5-2007

A Formal Approach to Specifying and Verifying Spacecraft Behavior

Allan I.S. McInnes
Utah State University

Follow this and additional works at: https://digitalcommons.usu.edu/etd

Part of the Electrical and Computer Engineering Commons

Recommended Citation
https://digitalcommons.usu.edu/etd/268

This Dissertation is brought to you for free and open access by the Graduate Studies at DigitalCommons@USU. It has been accepted for inclusion in All Graduate Theses and Dissertations by an authorized administrator of DigitalCommons@USU. For more information, please contact digitalcommons@usu.edu.
Abstract

A Formal Approach to Specifying and Verifying Spacecraft Behavior

by

Allan I. S. McInnes, Doctor of Philosophy
Utah State University, 2007

Major Professor: Dr. Charles M. Swenson
Department: Electrical and Computer Engineering

Process algebra can provide spacecraft designers with a mathematical formalism for specifying, understanding, analyzing, and verifying spacecraft system behavior. Although it is standard practice to mathematically model and analyze the subsystems of a spacecraft to ensure that they will function correctly when built, the system-level behavior of the spacecraft is generally understood in much less rigorous terms. This leaves the spacecraft system vulnerable to design errors which may not become apparent until the integration and test phase, when design changes are most expensive. In this dissertation, we develop a formal approach to engineering spacecraft behavior, based on mathematical models of behavior expressed using the process algebra Communicating Sequential Processes. This new approach to spacecraft behavior is intended to help spacecraft systems engineers to model and analyze proposed spacecraft system designs in a rigorous manner, and to detect subtle specification and design errors earlier in the design process than the errors would otherwise be found.

(309 pages)
To Mum and Dad. They’ve always believed I could do it, whatever the *it* was.
Acknowledgments

Completing this dissertation has been a far more difficult, and yet rewarding, experience than I ever imagined it would be when I first set out to obtain a Ph.D. It’s safe to say that I wouldn’t have even come close to reaching that goal without the support and encouragement of a lot of different people (and several coffee-houses).

First and foremost, I’d like to thank my advisor, Dr. Charles Swenson, for getting me started on this path in the first place, and for keeping me on it once I was there. Chuck has been willing to let me pursue my own wild ideas on spacecraft systems engineering (giving me enough rope to hang myself with?), for which I’m very grateful.

The other members of my committee have also been an invaluable part of my graduate experience. Dr. Dyke Stiles introduced me to CSP, fielded my many questions on the finer points of CSP semantics, proof-read early drafts of some of the chapters contained herein, and helped to introduce me to the wider CSP community. This dissertation obviously could not exist without him. Dr. Brandon Eames has been a great collaborator (co-conspirator?), over the past year or so. I’ve enjoyed our wide-ranging discussions on modeling languages, embedded systems, and the joys of programming in Mozart/Oz. Dr. Rees Fullmer has been as much a friend as a committee member, and always good for an interesting conversation, or a discussion of the importance of coffee. Dr. YangQuan Chen has provided ongoing encouragement for me to finish my degree: I can’t count the number of times I’ve run into Dr. Chen in the corridor, and had him express surprise that I was still there. I’d also like to thank Dr. Kevin Moore and Dr. Annette Bunker, both former members of my committee, for their influence on my graduate career.

My Ph.D. studies from August 2003 through August 2006 were generously supported through a Space Dynamics Laboratory Tomorrow Ph.D. Fellowship. I am extremely grateful to SDL for awarding me the fellowship, and wish to extend special thanks to Dr. Steve Hansen and Dr. Pat Patterson for taking a continuing interest in my work and personal well-being.
My friends around USU, Aroh, Dave, Todd, Mike, Mary, Justin, Laurie, Cynthia, Megan, and Jeff K, somehow managed to make Logan a fun place to be. I wish more of you could have stayed in Logan the whole 3 years I was there. But I’m thankful for every moment that you were around.

Erin gave me endless moral support and encouragement through the majority of my work on this dissertation. I’m sorry that she wasn’t able to see this through to the end with me, but grateful for the support she gave me.

In Logan, Caffe Ibis was my home away from home. In Boulder, the cast and crew of the Folsom St. Coffee Co. have probably started to wonder if I’m bolted to the floor. Thanks to the staff and regulars at both coffee-houses.

It’s funny how Leland and I always seem to end up working on graduate degrees at the same time. I’m not sure why that happens. The important thing was that we both went through this at the same time, and could lean on each other when we needed to. Or just have a long-winded conversation about some esoteric aspect of software engineering, if that was more what we needed.

Last, but by no means least, I want to thank Mum and Dad. For everything. You brought me into this world, made me who I am, and have supported me through everything I’ve ever done, or tried to do. I know just how lucky I am, and I’m eternally grateful that you are my parents. Thank you.

Allan McInnes
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>List of Tables</td>
<td>x</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xi</td>
</tr>
<tr>
<td>Notation</td>
<td>xiv</td>
</tr>
<tr>
<td>Acronyms</td>
<td>xviii</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>2</td>
</tr>
<tr>
<td>1.2 State of the Art</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Approach</td>
<td>5</td>
</tr>
<tr>
<td>1.4 Plan of the Dissertation</td>
<td>7</td>
</tr>
<tr>
<td>2 Related Work</td>
<td>9</td>
</tr>
<tr>
<td>2.1 Formal Methods</td>
<td>9</td>
</tr>
<tr>
<td>2.2 CSP</td>
<td>10</td>
</tr>
<tr>
<td>2.2.1 Recent Research</td>
<td>11</td>
</tr>
<tr>
<td>2.2.2 Practical Applications</td>
<td>13</td>
</tr>
<tr>
<td>2.3 Formalizing Aspects of Spacecraft Behavior</td>
<td>14</td>
</tr>
<tr>
<td>2.3.1 CSP</td>
<td>15</td>
</tr>
<tr>
<td>2.3.2 Promela/SPIN</td>
<td>16</td>
</tr>
<tr>
<td>2.3.3 Other Formalisms</td>
<td>17</td>
</tr>
<tr>
<td>2.4 Summary</td>
<td>18</td>
</tr>
<tr>
<td>3 The Process Algebra CSP</td>
<td>19</td>
</tr>
<tr>
<td>3.1 CSP Notation</td>
<td>20</td>
</tr>
<tr>
<td>3.1.1 Events</td>
<td>21</td>
</tr>
<tr>
<td>3.1.2 Prefixing and Recursion</td>
<td>21</td>
</tr>
<tr>
<td>3.1.3 Parallel Composition</td>
<td>22</td>
</tr>
<tr>
<td>3.1.4 Alternative Composition</td>
<td>25</td>
</tr>
<tr>
<td>3.1.5 Channels</td>
<td>26</td>
</tr>
<tr>
<td>3.1.6 Sequential Composition</td>
<td>28</td>
</tr>
<tr>
<td>3.1.7 Hiding</td>
<td>29</td>
</tr>
<tr>
<td>3.1.8 Scoping</td>
<td>30</td>
</tr>
<tr>
<td>3.2 CSP Theory</td>
<td>31</td>
</tr>
<tr>
<td>3.2.1 Denotational Models</td>
<td>31</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Summary of guidelines for channel and type construction. 107</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Cartoon of a process as a black box with two events in its interface.</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>Transition system for the <em>Days</em> process.</td>
<td>22</td>
</tr>
<tr>
<td>3.3</td>
<td>Transition system for the <em>CommSystem</em> process.</td>
<td>23</td>
</tr>
<tr>
<td>3.4</td>
<td>The parallel composition <em>ControlledThruster</em>.</td>
<td>24</td>
</tr>
<tr>
<td>3.5</td>
<td>External choice in <em>Transceiver</em> viewed as a “box with buttons.”</td>
<td>26</td>
</tr>
<tr>
<td>3.6</td>
<td>Transition system for the <em>CopyBits</em> process.</td>
<td>27</td>
</tr>
<tr>
<td>3.7</td>
<td>Transition system for the <em>MoreBits</em> process.</td>
<td>28</td>
</tr>
<tr>
<td>3.8</td>
<td>Communicating through channels.</td>
<td>29</td>
</tr>
<tr>
<td>3.9</td>
<td>Hidden channel communications.</td>
<td>30</td>
</tr>
<tr>
<td>3.10</td>
<td>Transition system for $P \parallel Q$.</td>
<td>43</td>
</tr>
<tr>
<td>4.1</td>
<td>Structure of the composite <em>Science</em> process.</td>
<td>56</td>
</tr>
<tr>
<td>4.2</td>
<td>Example of an event sequence.</td>
<td>57</td>
</tr>
<tr>
<td>4.3</td>
<td>Mode transition diagram for a simple spacecraft.</td>
<td>58</td>
</tr>
<tr>
<td>4.4</td>
<td>Requirement involving state transitions.</td>
<td>58</td>
</tr>
<tr>
<td>4.5</td>
<td>Example of a triggered behavior requirement.</td>
<td>61</td>
</tr>
<tr>
<td>4.6</td>
<td><em>Between</em> and <em>Outside</em> constraints.</td>
<td>71</td>
</tr>
<tr>
<td>4.7</td>
<td>EO-1 spacecraft behavior in different modes.</td>
<td>73</td>
</tr>
<tr>
<td>4.8</td>
<td>Graphical elements of FFBD notation.</td>
<td>75</td>
</tr>
<tr>
<td>4.9</td>
<td>Simple example of an FFBD.</td>
<td>78</td>
</tr>
<tr>
<td>4.10</td>
<td>Example of an event timing requirement.</td>
<td>79</td>
</tr>
<tr>
<td>5.1</td>
<td>Example of a spacecraft system block diagram.</td>
<td>91</td>
</tr>
</tbody>
</table>
5.2 Illustration of structural similarity between a block diagram and a process model. ................................................................. 94
5.3 Power subsystem interfaces for the ACE spacecraft. ................. 95
5.4 Examples of discrete signals on the ACE spacecraft. .................. 96
5.5 ACE power subsystem commands. ...................................... 97
5.6 Telemetry for the WIRE array deployment mechanism. ............... 99
5.7 Abstract states Attitude 1 and Attitude 2 represent qualitatively different attitudes. ................................................................. 99
5.8 Connectivity of the systembus channel. .................................. 102
5.9 Example of an implicit interface on the WIRE spacecraft. .......... 104
5.10 Explicit power interface, and implicit attitude interface. .......... 105
5.11 Example of abstract specification events. ............................. 106
5.12 Assumed behavior of the MGS attitude control system. .......... 113
5.13 The simple ADCS model of example 5.4.2.1. ......................... 115
5.14 The ADCS model of example 5.4.2.2. ................................ 118
5.15 EO-1 spacecraft ADCS modes and transitions. ....................... 119
5.16 ADCS mode transitions for the model discussed in example 5.4.2.3. 121
5.17 The ADCS model of example 5.4.2.3. ................................ 124
5.18 The ADCS model of example 5.4.2.4. ................................ 126
5.19 Command distribution requirement for the WIRE CDH. .......... 134
5.20 Specification of stored-command handling for WIRE. ............... 136
5.21 Controlling the execution of stored sequence processes. .......... 137
5.22 Modeling sequence-loading using tokens. ............................ 138
5.23 Specification of data storage modes for ACE. ....................... 143
5.24 Specification of data-handling for the WIRE spacecraft. ......... 143
5.25 The EPS model of example 5.4.4.1. .................................. 153
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.26</td>
<td>The EPS model of example 5.4.4.3.</td>
<td>156</td>
</tr>
<tr>
<td>5.27</td>
<td>EPS switch-command handling for the WIRE spacecraft.</td>
<td>158</td>
</tr>
<tr>
<td>5.28</td>
<td>Power buses on the WIRE spacecraft.</td>
<td>159</td>
</tr>
<tr>
<td>5.29</td>
<td>Payload operation requirements for the WIRE CDH.</td>
<td>166</td>
</tr>
<tr>
<td>5.30</td>
<td>Example schematic for a monopropellant blowdown propulsion subsystem.</td>
<td>171</td>
</tr>
<tr>
<td>5.31</td>
<td>System block diagram corresponding to the CSP in example 5.5.1.1.</td>
<td>183</td>
</tr>
<tr>
<td>5.32</td>
<td>Message sequence chart for the scenario described in example 5.5.5.1.</td>
<td>192</td>
</tr>
<tr>
<td>5.33</td>
<td>Verifying properties of a spacecraft system model using FDR.</td>
<td>200</td>
</tr>
</tbody>
</table>
Notation

Sets and Logic

¬x  negation - not x
x ∧ y  conjunction - x and y
x ∨ y  disjunction - x or y
x ⇒ y  implication - ¬x ∨ y
x ⇔ y  double implication - (x ⇒ y) ∧ (y ⇒ x)
∀x • P  P holds for all x (universal quantification)
∃x • P  P holds for at least one x (existential quantification)
∅  the empty set
{x₁, . . . , xₙ}  set containing the elements x₁, . . . , xₙ
#X  cardinality of X
x ∈ X  x is a member of the set X
X ⊆ Y  X is a subset of Y - ∀x • x ∈ X ⇒ x ∈ Y
X ∪ Y  union
X ∩ Y  intersection
X \ Y  set difference
{a | P}  set comprehension - the set of a such that P holds
X × Y  Cartesian product - {(a, b) | a ∈ X ∧ b ∈ Y}
PX  power set (i.e. the set of all subsets) of X.

Events

α(P)  set of all events in which process P engages (alphabet)
Σ  set of all events (universal alphabet)
✓  successful termination signal
τ  invisible (internal) event
a.b.c  compound event
datatype \( T = s_1 \mid s_2 \mid \cdots \) declaration of type \( T \) composed of symbols \( s_1, s_2, \cdots \)

nametype \( N = \cdots \) declaration of a type \( N \) composed of symbols drawn from previously declared datatypes

channel \( c : T \) declaration of channel \( c \) of type \( T \)

\( c?x \) input on channel \( c \)

\( c!x \) output on channel \( c \)

\( \{ c \} \) set of all events associated with channel \( c \)

Processes

\( a \rightarrow P \) prefixing

\( P \parallel Q \) interleaved parallel

\( P \| [A | B] \parallel Q \) alphabetized parallel

\( P \| [X] \parallel Q \) interface parallel

\( P \sqcap Q \) external choice

\( P \sqcap Q \) nondeterministic choice

\( P ; Q \) sequential composition

if \( b \) then \( P \) else \( Q \) conditional

\( b \& P \) boolean guard

\( P \setminus X \) hiding

\( P \triangle Q \) interrupt

\( P[x : A \bullet b \leftarrow c.\downarrow x] \) renaming of events \( \{ b \} \) to events \( \{ c.\downarrow x \} \) where \( x \in A \)

\( \square a : A \bullet P(a) \) generalized external choice

\( \| i : I \bullet A(i) \circ P(i) \) generalized parallel

\( ; i : I \bullet P(i) \) generalized sequential composition

let \( x = y \) within \( P \) local definition

\( P \sqsubseteq_M Q \) refinement under the semantic model \( M \)

\( \mathcal{L}_A(P) \) lazy abstraction of events in \( A \)
Traces

\(\emptyset\) empty sequence

\(\langle a_1, \ldots, a_n \rangle\) sequence \(a_1, \ldots, a_n\), in that order

\(s_1 \triangleleft s_2\) \(s_1\) concatenated with \(s_2\), e.g. \(\langle a \rangle \triangleleft \langle b \rangle = \langle a, b \rangle\)

\(s_1 \leq s_2\) \(s_1\) is a prefix of \(s_2\), e.g. \(\langle a \rangle \leq \langle a, b \rangle\)

\(s \setminus X\) hiding - all members of \(X\) removed from \(s\)

\(s \uparrow X\) restriction - hide all but members of \(X\)

\(s \downarrow c\) sequence of values in \(s\) communicated over channel \(c\)

\(s \downarrow a\) number of \(a\) events in \(s\)

\(\text{head}(s)\) first element of \(s\), e.g. \(\text{head}(\langle a \rangle \triangleleft s') = a\)

\(\text{tail}(s)\) sequence obtained by removing head, e.g. \(\text{tail}(\langle a \rangle \triangleleft s') = s'\)

\(\text{last}(s)\) last element of \(s\), e.g. \(\text{last}(s' \triangleleft \langle a \rangle) = a\)

Machine-Readable CSP (CSP\(_M\))

Sets and Logic

\(\neg x\) negation

\(x \land y\) conjunction

\(x \lor y\) disjunction

\(\{}\) the empty set

\(\text{member}(x, X)\) \(x\) is a member of the set \(X\)

\(\text{union}(X, Y)\) union

\(\text{inter}(X, Y)\) intersection

\(\text{diff}(X, Y)\) set difference

\(\{a \mid P\}\) set comprehension - the set of \(a\) such that \(P\) holds

Sequences

\(\langle\rangle\) empty sequence

\(\langle a_1, \ldots, a_n \rangle\) sequence \(a_1, a_2, a_3, \ldots\), in that order

\(s_1 \triangleleft s_2\) concatenation

\(\text{head}(s)\) first element of \(s\)
tail(s) sequence obtained by removing head

Events

Events set of all events (universal alphabet)
a.b.c compound event
datatype T = s1 | s2 | ... datatype declaration
nametype N = ... nametype declaration
channel c : T channel declaration
c ? x input
c ! x output
{|c|} set of all events associated with channel c

Processes

a -> P prefixing
P ||| Q interleaved parallel
P [ A||B ] Q alphabetized parallel
P [| X |] Q interface parallel
P [] Q external choice
P |~| Q nondeterministic choice
P ; Q sequential composition
if b then P else Q conditional
b & P boolean guard
P \ X hiding
P /\ X interrupt
P [[ b <- c.x | x <- A ]] renaming
[ ] a:A @ P(a) generalized external choice
|| i:I @ [A(i)] P(i) generalized parallel
let x = y within P local definition
P [M= Q refinement under the semantic model M
### Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACE</td>
<td>Advanced Composition Explorer</td>
</tr>
<tr>
<td>ACP</td>
<td>Algebra of Communicating Processes</td>
</tr>
<tr>
<td>ADCS</td>
<td>Attitude Determination and Control Subsystem</td>
</tr>
<tr>
<td>AOCS</td>
<td>Attitude and Orbit Control Subsystem</td>
</tr>
<tr>
<td>CCS</td>
<td>Calculus of Communicating Systems</td>
</tr>
<tr>
<td>CDH</td>
<td>Command and Data Handling Subsystem</td>
</tr>
<tr>
<td>CSP</td>
<td>Communicating Sequential Processes</td>
</tr>
<tr>
<td>CSP$_M$</td>
<td>Machine-Readable Communicating Sequential Processes</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain-Specific Language</td>
</tr>
<tr>
<td>EO-1</td>
<td>Earth Observing 1</td>
</tr>
<tr>
<td>EPS</td>
<td>Electrical Power Subsystem</td>
</tr>
<tr>
<td>FDR</td>
<td>Failures-Divergence Refinement</td>
</tr>
<tr>
<td>FFBBD</td>
<td>Functional Flow Block Diagram</td>
</tr>
<tr>
<td>FSW</td>
<td>Flight Software</td>
</tr>
<tr>
<td>GSFC</td>
<td>Goddard Space Flight Center</td>
</tr>
<tr>
<td>HESSI</td>
<td>High Energy Solar Spectroscopic Imager</td>
</tr>
<tr>
<td>ICD</td>
<td>Interface Control Document</td>
</tr>
<tr>
<td>JHUAPL</td>
<td>Johns Hopkins University Applied Physics Laboratory</td>
</tr>
<tr>
<td>JPL</td>
<td>Jet Propulsion Laboratory</td>
</tr>
<tr>
<td>LTL</td>
<td>Linear Temporal Logic</td>
</tr>
<tr>
<td>MBED</td>
<td>Model-Based Engineering Design</td>
</tr>
<tr>
<td>MGS</td>
<td>Mars Global Surveyor</td>
</tr>
<tr>
<td>MIMO</td>
<td>Multiple-Input Multiple-Output</td>
</tr>
<tr>
<td>NASA</td>
<td>National Aeronautics and Space Administration</td>
</tr>
<tr>
<td>OMT</td>
<td>Object Modeling Technique</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>ProBE</td>
<td>Process Behavior Explorer</td>
</tr>
<tr>
<td>PVS</td>
<td>Prototype Verification System</td>
</tr>
<tr>
<td>SACI-1</td>
<td>Satélite Científico 1</td>
</tr>
<tr>
<td>SCR</td>
<td>Software Cost Reduction</td>
</tr>
<tr>
<td>SpecTRM</td>
<td>Specification Tools and Requirements Methodology</td>
</tr>
<tr>
<td>SysML</td>
<td>Systems Modeling Language</td>
</tr>
<tr>
<td>TLA+</td>
<td>Temporal Logic of Actions</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHSIC Hardware Description Language</td>
</tr>
<tr>
<td>VHSIC</td>
<td>Very-High-Speed Integrated Circuit</td>
</tr>
<tr>
<td>WIRE</td>
<td>Wide-Field Infrared Explorer</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

“Correctness is clearly the prime quality.
If a system does not do what it is supposed to do,
then everything else about it matters little.”
– Bertrand Meyer

Process algebra can provide spacecraft designers with a mathematical formalism for specifying, understanding, analyzing, and verifying spacecraft system behavior.

Behavior can be informally defined as “the order in which a system does things.” For example, the behavior of a dynamical system such as a simple pendulum can be described by a set of differential equations that define the sequences of positions and velocities which the pendulum can be observed to move through, and how those sequences change in response to the external inputs. Similarly, in describing spacecraft system behavior, we are interested in defining the sequences of things the spacecraft can be observed to do, and how those sequences change in response to different inputs.

In the broadest sense, the term “spacecraft system behavior” could encompass a complete description of the evolution of the state of a spacecraft, including the continuous dynamics that describe the evolution of the spacecraft position, orientation, and structural characteristics. However, in this dissertation we confine ourselves to considering the discrete, event-driven dynamics of the spacecraft system. We restrict ourselves in this way for two reasons:

1. Event-driven behavior such as commanded system reconfigurations and autonomous fault-responses can be complex, subtle, and mission-critical, but has received very little study compared to continuous spacecraft dynamics.
The operationally important aspects of spacecraft behavior are typically either discrete events, or qualitative changes in continuous behavior that can be treated as discrete events.

We use the process algebra *Communicating Sequential Processes* (CSP) [1–3] as a vehicle for demonstrating the thesis that process algebra can provide spacecraft designers with a mathematical formalism for specifying, understanding, analyzing, and verifying spacecraft system behavior. CSP, like all process algebras [4], is a tool for describing the behavior of systems made up of components that interact through an exchange of discrete messages, and in which the history of past communications influences the future behavior of the system. By viewing spacecraft systems as collections of interacting component subsystems, we can make use of the concepts embodied in CSP to enrich our understanding of spacecraft system behavior. This dissertation introduces a collection of definitions and guidelines for developing formal mathematical descriptions of spacecraft system behavior using CSP, and demonstrates their use on several different example specifications. The definitions and guidelines that we introduce provide a toolkit for mathematically specifying, reasoning about, simulating, and analyzing spacecraft system behavior, and a foundation for further research into spacecraft system behavior.

1.1 Motivation

Current approaches to spacecraft system design have been used to produce many successful spacecraft. However, those approaches do not provide for verification of the correctness of designed spacecraft system behaviors prior to the completion of the detailed system design. As a result, spacecraft developed using current approaches are highly vulnerable to design errors that cannot be detected until the integration and test phase, when design changes are most expensive in terms of cost and schedule. Disconnects between required and designed spacecraft system behaviors have resulted in development difficulties and cost overruns in several major missions [5].
In contrast to the current practice in spacecraft system behavior design, most engineering design efforts rely on mathematical theories and models that allow engineers to precisely specify a design, to reason about and simplify the design, and to analyze the design to ensure that it will meet its requirements [6, 7]. For example, during the spacecraft design process, attitude control engineers develop mathematical models of the dynamics of the spacecraft and the proposed control laws [8], and structural engineers develop and analyze finite element models of the proposed structural configuration [9]. In his illuminating investigation of the nature of engineering knowledge, What Engineers Know and How They Know It, Walter Vincenti points out that

“although engineering activity produces artifacts, conceiving and analyzing these artifacts requires thoughts in people’s minds; the clearer these thoughts, the more likely it is that the artifacts will be successful.” [10]

Spacecraft designers and systems engineers inherently have internal mental models of the behavior they expect of a spacecraft system. By providing a mathematical framework within which to understand the behavior of spacecraft systems, those internal mental models can be made clearer and more precise. Moreover, given a mathematical framework for modeling, internal mental models can be made explicit, external, and subject to rigorous analysis. The transition to formal mathematical methods in the traditionally informal discipline of software engineering has been found to significantly reduce costly requirements errors and integration problems [11–14], to help uncover ambiguities, incoherences, and errors in a design much earlier than they would otherwise be apparent [15], and to aid in the understanding of complex systems [16–18].

1.2 State of the Art

Current spacecraft design practices do not include the creation of formal, mathematical models of spacecraft system behavior in any phase of the development process.

Although establishing the desired behavior of a spacecraft is considered a key part of the initial mission concept definition process [19], the primary tool used to define that behav-
ior is informal text, sometimes accompanied by informal diagrammatic descriptions such as functional flow block diagrams [20, 21]. When we examine typical spacecraft system requirements and specification documents, such as the *ACE Spacecraft Design Specification* [22], the *MGS Spacecraft Requirements* [23], and the *WIRE System Requirements* [24], we find that they contain only limited information on required system-level behavior. Where behavior is explicitly defined, the specifications are usually expressed informally, using natural language statements, and are often fragmentary, ambiguous, and incomplete.

At the conceptual level, spacecraft design tends to focus more on defining the physical architecture of the spacecraft, and estimating static quantities such as system mass and power budgets, than on defining system behavior [25, 26]. In those cases where dynamic system-level spacecraft models are developed, they typically focus on estimating resource consumption for simple mission profiles [27, 28], rather than considering command-driven and data-driven spacecraft behavior. Popular textbooks on spacecraft systems engineering, such as Fortescue and Stark [29], Pisacane and Moore [30], or Wertz and Larson [19], provide detailed discussions of mathematical techniques for modeling and analyzing individual spacecraft subsystems, but contain little or no information on analyzing system-level behavior, and make no mention of mathematical methods for analyzing such behavior.

As spacecraft development progresses, isolated aspects of spacecraft behavior may undergo rigorous definition and analysis, particularly in areas such as fault-protection [31] or autonomous control [32]. However, these efforts, if they are undertaken, are largely isolated to the flight software component of the spacecraft system. Spacecraft designers primarily rely on experience, discipline, and extensive peer reviews to detect errors in spacecraft behavior up until the point where the design is well-defined enough to conduct hardware-in-the-loop tests. Designs often must undergo lengthy test/debug cycles during system integration and testing to discover and repair unexpected subsystem interactions [33, 34].

The problems created by the lack of tools to support early verification of spacecraft system designs have motivated a number of researchers to explore ways of detecting design errors earlier in the process. Several researchers have investigated formal approaches to
designing and verifying certain aspects of spacecraft behavior [31, 35–37]. However, these researchers have focused on software verification rather than system design. Some headway has recently been made on enabling system behavior validation to begin prior to hardware-in-the-loop testing, by using a simulated “virtual spacecraft” model to provide a runtime environment for testing and validating onboard software [38]. However, the simulations involved require a significant amount of design and development work to be completed before the simulation can be used, and are again focused on software verification rather than system design. The Model-Based Engineering Design (MBED) initiative at NASA’s Jet Propulsion Laboratory is an attempt to use modeling and simulation throughout the spacecraft specification and design process, and includes some consideration of system behavior [5]. However, MBED has a strong resource-oriented bias, and uses techniques that do not provide the rich set of mathematical tools associated with process algebras.

1.3 Approach

The classical mathematical tool for describing the dynamical behavior of a system is the differential equation. However, differential equations are not appropriate for describing the kind of event-driven behavior that we are interested in examining [39].

Instead of using differential equations to describe event-driven spacecraft behavior, we abstract spacecraft behavior into CSP processes composed of sequences of discrete events. These events can obviously represent things that are usually considered discrete, such as software state changes, or components being switched from off to on. But discrete events can also be used to represent important qualitative changes in things that are usually treated as continuous, such as a change in the attitude of a spacecraft from the small angular rates of a nominally fixed attitude to the large angular rates of a slew maneuver. This abstraction of qualitative changes into discrete events is analogous to the way in which a dynamicist may abstract a complex shape into a simpler geometry, for example treating a planet as a point mass, in order to make an analysis of its dynamics more tractable. Such abstractions necessarily suppress some details of the system being modeled, but can be used to obtain useful results that might otherwise be infeasible to compute.
Several formalisms are available for reasoning about discrete-event systems. In addition to simple familiarity, we have several reasons for choosing to use CSP as our tool for formalizing spacecraft behavior:

- CSP is a rich mathematical theory of concurrency, and provides a powerful set of tools upon which to build an understanding of spacecraft behavior, including theoretical results in areas such as buffering [3], deadlock-free design [40], and data independence [41,42].

- CSP supports a mathematical notion of refinement, which provides a convenient way for expressing relationships between models defined at different levels of abstraction.

- CSP has a long history of practical application and industrial use in areas such as software design and synthesis [43], hardware design [44], fault-tolerant and dependable systems design [45,46], and security protocol verification [47]. This fact increases our confidence that the techniques we develop can eventually be transitioned into industry.

- Industrial tool support for CSP-based design and analysis is available in the form of the FDR model-checker [48] and the ProBE process animator [49], both produced by Formal Systems (Europe) Ltd.

In short, CSP provides a good combination of theoretical depth and practical application.

Our basic approach for developing a CSP-based formalization of spacecraft system behavior consists of the following steps:

1. Examine articles, textbooks, and documentation on spacecraft requirements and design, and identify informal behavior-related concepts embodied in those resources.

2. Develop a set of principles for mapping previously identified informal behavior concepts onto CSP constructs such as datatypes, channels, and process expressions.

3. Build models based on the principles developed in the preceding step.

4. Experiment with the models, and use the lessons learned in those experiments to refine the principles upon which the models are built.
We consider models of spacecraft behavior at two different levels of abstraction: black-box specifications of desired system behavior, and block-diagram-level models of the behavior of interacting spacecraft subsystems. The former type of model represents the required spacecraft behavior, but does not specify the mechanism by which that behavior is produced. The latter type of model represents a proposed spacecraft design, the subsystems of which, when interacting in the prescribed way, should produce the behavior specified by a black-box behavior model. We use a refinement relation to express the relationship between a system behavior specification and a system design model, and verify the refinement relation to confirm that the proposed design produces the required behavior.

1.4 Plan of the Dissertation

In the remainder of this dissertation we demonstrate that the process algebra CSP can provide spacecraft designers with a mathematical formalism for specifying, understanding, analyzing, and verifying spacecraft system behavior. This work is arranged into the following chapters:

**Related Work.** In which we present a survey of existing work that is related to the research contained herein. The survey includes an overview of some of the previous research on new approaches to spacecraft behavior design that were briefly mentioned in section 1.2.

**The Process Algebra CSP.** In which we introduce the basic theory and notation of CSP, and discuss the application of CSP to specifying and verifying systems.

**Specifying Spacecraft System Behavior.** In which we develop a conceptual framework within which to understand and specify black-box spacecraft system behavior. Included within the development of this framework are mappings from several informal notations and concepts commonly used in discussing spacecraft behavior, to CSP processes.

**Modeling Spacecraft Subsystem Interactions.** In which we develop a collection of guidelines for creating process models of each spacecraft subsystem, and connecting those models into a spacecraft system model that is structurally similar to a spacecraft system block diagram. The chapter also includes a small catalog of different kinds of properties
that a spacecraft model may be verified to possess, and provides several examples of the verification of these properties.

**Conclusions.** In which we summarize the work presented in this dissertation, identify the specific contributions embodied in this work, and discuss directions for future research that builds upon our work.

**Appendices.** In which we provide the complete CSP text of the library of specification constructs developed as part of the work reported in this dissertation, as well as the complete text of several lengthy specification examples referred to in the main body of this dissertation.
Chapter 2
Related Work

“Taken out of context, I must seem so strange.”
– Ani DiFranco

This chapter provides a context for the research described in this dissertation by discussing related work. First, we discuss the family of mathematical approaches to the design and analysis of computer software and hardware, collectively known as “formal methods,” of which process algebras such as CSP are a member. We then briefly survey recent academic work on CSP, as well as various examples of applying CSP to practical problems in industry. Finally, we review existing work on applying formal methods to different aspects of spacecraft behavior.

2.1 Formal Methods

Systems composed of software and digital hardware typically exhibit discrete dynamic behavior that is difficult to express using the continuous dynamical models traditionally used in engineering analysis. The term formal methods encompasses a variety of mathematical techniques that have been developed over the past 40 years to enable engineers to rigorously specify, model, and reason about software and hardware systems which have a discrete state-space, and discontinuous dynamics. Although formal methods vary in the aspects of a system which they emphasize or are suitable for analyzing, the common foundations of most formal methods are mathematical logic and discrete mathematics [50]. Similar ideas have recently become popular in control engineering circles, where they are referred to as discrete event control theory [51].

Using formal methods, the specification or design for a software or hardware system can be mathematically modeled. The resulting model can be rigorously analyzed for internal
consistency, or for conformance to a desired set of properties or behaviors. Such formal verification of specifications and designs can take the form of deductive logical proofs, often carried out using proof assistants such as Isabelle/HOL or PVS [52], or may be performed through exhaustive state exploration techniques known as model-checking [53].

The use of formal methods during the design process has been found to improve understanding of the requirements and design of a system, and to enhance communication between design team members [17]. Improved understanding and communication allows specification and design errors to be caught earlier in the design cycle. Earlier detection of design errors has been observed to have several impacts, including reduced rework rates, lower testing costs due to more systematic test definition [17], and up to an order of magnitude reduction in defect rates in delivered software and hardware [54]. Successful industrial applications of formal methods have included the avionics software for the Lockheed C130J, the Federal Aviation Administration’s second generation Traffic Collision Avoidance System (TCAS), and the IEEE FutureBus+ cache coherency protocol [54].

The benefits associated with the mathematical formalization of discrete systems have prompted several researchers to investigate how formal methods could be used outside of their traditional applications areas. For example, process algebras have recently been applied to such diverse fields as systems biology [55], and business process modeling [56], in the hope that a formalization of these areas will lead to improvements in both the understanding of existing systems, and the specification of new systems.

### 2.2 CSP

Within the larger formal methods community, several approaches have been developed specifically for reasoning about systems involving interacting elements, including concurrent, parallel, distributed, and reactive systems. An incomplete list of these formalisms includes: Petri Net theory [57], the Actors model [58], Temporal Logic of Actions (TLA+) [59], and I/O Automata [60], along with process algebras such as the Calculus of Communicating Systems (CCS) [61], the π-Calculus [62], the Algebra of Communicating Processes (ACP) [63], and, of course, CSP.
CSP was originally developed to allow formal reasoning about complex concurrent computer systems. Brookes et al. introduced the mathematical theory of CSP in a 1984 paper titled “A Theory of Communicating Sequential Processes” [1]. Since that first publication a great deal of work has been done to extend the original theory [3]. Commercial tools such as the FDR model-checker have become available [48], and enabled widespread practical use of CSP for software design, digital hardware design, and formal verification of models developed in informal notations.

2.2.1 Recent Research

Recent research in extending CSP has both theoretical and practical aspects. On the theoretical side, work by Roscoe, Lazić, and Creese on data-independence techniques [41,42] appears to hold a great deal of promise for extending the applicability of state-exploration methods to arbitrarily large systems. Ouaknine has explored the relationship between continuous time and discrete time variants of CSP, and demonstrated that continuous time specifications can successfully be translated into a discrete time form suitable for model-checking [64]. Other recent theoretical work includes Martin’s development of tools and techniques to facilitate automated proofs in Hoare’s original denotational style [65], and Roscoe’s investigations into the types of specifications that can feasibly be expressed using the modern process refinement specification style [66]. This research is significantly expanding the range and size of systems to which CSP can be successfully applied.

Research into increasing the practicality of applying CSP to industrial problems includes Zhou’s work on the development of tools for automated manipulation of CSP process expressions [67], Phillips’ tool for automated conversion of CSP specifications into hardware [44], Raju’s automated software generation tool [68], work by Kundu et al. on a CSP refinement-checker capable of checking systems with infinite state-spaces [69], and work by Peleska on using CSP to define test specifications [70]. In an effort to make CSP more accessible, Hilderink has recently proposed a new, formal graphical notation based on CSP [71]. The use of this new notation is not yet widespread. However, there is limited tool support available in the form of a graphical editor which can be used to construct
process models, and which has the capability to export machine-readable CSP suitable for model-checking using FDR [72].

A number of researchers have sought to use CSP to create a rigorous basis for reasoning about and analyzing otherwise informal diagrams and graphical notations. For example, Allen and Garlan developed the CSP-based Wright architectural description language in order to formalize the existing ad hoc “box and lines” diagramming methods used to define software architectures [18]. Allen has applied Wright to several case studies, including the U.S. Navy’s AEGIS weapons system, and the Department of Defense’s High Level Architecture for Simulation. These case studies demonstrated the usefulness of a formal architectural description for exposing ambiguities and omissions in specifications, detecting design errors, simplifying designs, and improving confidence in the implementation.

Roscoe has investigated formalizing Harel’s Statechart notation [73] in terms of CSP [74]. In addition to making specifications expressed in the Statechart notation amenable to efficient formal verification using the FDR model-checker, Roscoe’s work also facilitates the proof of general properties of a specification which are independent of specific parameter settings. Similar work by Ng [75, 76] and Engels [77] has resulted in formal, CSP-based interpretations for a subset of the Unified Modeling language (UML) [78] notation for state diagrams, and tools for automatically converting from the UML to CSP. The formal interpretation of UML permits a design expressed in the UML to be verified correct and deadlock-free, and makes it possible to establish the equivalences between different representations of a design. Fischer has explored the application of a CSP derivative called CSP-OZ to the construction of a precise formal semantics for the structure diagrams used in the real-time extension of the UML [79].

The application of CSP to the formal specification of spacecraft behavior permits spacecraft designers to leverage existing CSP research, and to draw on the experience of the CSP community in applying CSP to practical problems. Recent work by Eames et al. seeks make the use of CSP more accessible to spacecraft designers, by providing graphical tools that implement some of the ideas presented in this dissertation [80].
2.2.2 Practical Applications

An early and important application of CSP was in the development of the INMOS T9000, a complex superscalar pipelined processor designed specifically to support large-scale multiprocessing [81]. CSP was heavily employed in verifying the correctness of both the processor pipeline, and the complex Virtual Channel Processor which managed off-chip communications for the processor. This work was the impetus for the development of the original FDR model-checker.

More recently, Abdallah and Damaj have used CSP to produce a verifiably correct hardware implementation of the IDEA cryptographic algorithm [82]. The use of CSP in this project also made it possible to rapidly explore several different design options, all provably able to meet the specifications. Peel and Javier have applied CSP to the problem of ensuring that FPGA designs produced by their occam-to-FPGA compiler are deadlock-free and conform to their specifications [83]. This permits specification of designs at a much higher level of abstraction than typical hardware description languages, while also providing greater assurance of implementation correctness. Wang and his colleagues have recently capitalized on the CSP roots of the Balsa language for handshake-based asynchronous logic design [84] to develop a tool for translating Balsa into CSP, enabling design verification tasks such as proofs of deadlock-freedom to be carried out using FDR [85].

In the realm of software, a relatively small-scale application of CSP is Welch and Martin’s CSP model of the previously informally defined threads-based concurrency primitives integral to the Java programming language [86]. This model was used to diagnose and correct a subtle, deadlock-inducing race condition in the JCSP concurrency library, which is built on top of the Java concurrency primitives. Larger-scale applications of CSP to software design have typically focused on dependable and safety-critical systems. For example, a team from the Bremen Institute for Safe Systems and Daimler-Benz Aerospace used CSP to verify that a software-based fault management system and avionics interface (consisting of some 23,000 lines of code) intended for use on the International Space Station was free of deadlock and livelock [45, 46]. This verification activity was able to uncover a
number of errors that were potentially undetectable using testing alone, and is claimed to have required only a quarter of the time of an equivalent informal effort. Another large-scale application (approximately 100,000 lines of code) was the development of software for a secure, high availability smart card Certification Authority by Praxis Systems International. Praxis used CSP to model their design, and to verify that it was secure and free of deadlock. The use of CSP enabled the company to uncover and correct significant flaws in their original design. The resulting system has been found to have much lower defect rates than comparable systems [43].

Since it is ideally suited to modeling systems that incorporate complex message exchanges, many practical applications of CSP have involved the verification of communications and security protocols. A key example in this application area is Lowe’s use of CSP and FDR to discover a subtle and previously unknown attack on the Needham-Schroeder public-key authentication protocol, and then to develop a corrected protocol able to defeat the attack [87]. Broadfoot and Lowe used the data-independence methods developed by Roscoe and Lazić to verify a stream authentication protocol which was previously thought to be intractable for model-checking [88]. Their success clearly demonstrates the benefits of having an active research community working on CSP.

The applications of CSP described above illustrate the flexibility and generality of the CSP approach, and demonstrate that CSP is capable of supporting industrial-scale modeling and verification. In addition to the applications already mentioned, CSP has been applied in ways much more directly related to the subject of this dissertation. We discuss those applications in the next section.

2.3 Formalizing Aspects of Spacecraft Behavior

Although the use of mathematical modeling during the spacecraft design process is hardly new, formal mathematics have not to date been applied to the design of event-driven spacecraft behavior. In fact, it is only relatively recently that there has been any interest in bringing formality to even that subset of behavior represented by the spacecraft flight software. In this section we review some of the recent work on using formal methods
in the development of flight software and onboard computers, which is also, by extension, work on formalizing some aspects of spacecraft behavior.

2.3.1 CSP

Process algebra, and in particular CSP, has previously been used for modeling and verifying some aspects of spacecraft behavior on at least two different projects: SACI-1, and Abrixas.

Mota and Sampaio [89] describe the effort to formally model and verify elements of the fault-tolerant onboard computer for the Brazilian SACI-1 micro-satellite using CSP-Z, a combination of CSP and the Z formal notation. The verification process was primarily focused on providing assurance that the onboard computer was free of deadlock. Verification was carried out by translating the CSP-Z model into pure CSP, which was then model-checked using FDR.

Schlingloff et al. [36] discuss the use of CSP to define automated tests for verifying embedded control software on the Abrixas spacecraft. Specifications of the expected behavior of the power and thermal control unit for the Abrixas satellite were written in CSP. These specifications were then used to drive an automated testing system. This approach radically improved test coverage, and uncovered a number of software errors and software/hardware incompatibilities that would not have otherwise been discovered.

These efforts demonstrated the feasibility of modeling some aspects of event-driven spacecraft behavior with CSP. However, the use of CSP on both projects was limited to verifying specific aspects of single subsystems. Neither project sought to explore the general problem of specifying spacecraft system behavior with which the work presented in this dissertation is concerned.

Recent work by Hinchey et al. [90] at the NASA Goddard Spaceflight Center explores the application of formal methods, including CSP, to specifying and verifying the emergent behavior of so-called swarm missions. Unlike the present work, the swarm research focuses on interactions between spacecraft, rather than the behavior of an individual spacecraft.
2.3.2 Promela/Spin

A number of experiments involving the process description language Promela, and its associated SPIN model-checker [91], have been carried out at NASA.

Feather [37] and Barltrop [31] describe verification and validation efforts for an advanced fault protection system developed at NASA’s Jet Propulsion Laboratory (JPL). The complexity of this system, which involved a networked interface between various spacecraft components, and the desire to reuse elements of the system in several spacecraft designs, made formal verification particularly attractive. The existing fault protection design was specified using Statecharts. Verification of the design was carried out by translating the Statecharts to Promela, and using SPIN to check various behavioral properties expressed in Linear Temporal Logic (LTL). The analysis with SPIN uncovered several areas where the fault protection system was improperly making assumptions about its environment.

Gluck and Holzmann [92] experimented with the use of Promela and SPIN for the verification of flight software. Their project involved post-flight analysis of the downlink packet handler and command sequence handler in the Deep Space 1 flight software. The verification process was able to identify a known error in the flight software, as well as a similar but previously unknown error, and a rare race condition that could have prevented sequenced commands from being executed. In a project related to Gluck and Holzmann’s, Visser et al. [93] developed Java Pathfinder, a tool for automatically extracting Promela/SPIN models from software written in Java. The Pathfinder tool was experimentally applied to model-checking parts of the Deep Space 1 flight software.

Havelund et al. [94] formally verified portions of the LISP implementation of the Remote Agent software by abstracting the software into Promela, and model-checking the resulting models against various LTL specifications. During the verification effort a major design flaw was uncovered, along with several minor errors that might otherwise not have been caught. It is interesting to note that the Deep Space 1 mission actually experienced an in-flight deadlock situation caused by a design flaw similar to the one detected in the Remote Agent, but located in a piece of software that had not been subjected to formal verification.
More recently, Smith et al. [32] developed techniques for translating the task and resource models used by a spacecraft autonomous planning systems into Promela. The resulting Promela models can be verified using SPIN to ensure that the corresponding planner model cannot produce undesirable plans.

There are two key differences between all of these efforts and the work presented in this dissertation. The most obvious difference is the use of the SPIN model-checker, and its associated Promela modeling language, rather than the process algebraic approach developed in this dissertation. More importantly, all of the experiments focus on the behavior of spacecraft software, while the present work considers spacecraft system behavior in general.

2.3.3 Other Formalisms

Easterbrook et al. [13] describe several experiments within NASA involving the application of formal methods to fault protection systems. Of the three case studies presented, two involved software for the International Space Station, which was specified and verified using a combination of the SCR specification method, and the PVS theorem prover. The third case study focused on verifying aspects of the fault protection software for the Cassini mission, using Object Modeling Technique (OMT) diagrams to clarify the existing textual requirements, and a mapping from OMT to the input language of the PVS prover to enable verification of behavioral properties. All of the experiments focused on the behavior of spacecraft software, rather than on the larger problem of spacecraft system behavior.

Weiss [95] and Ong [96] explore general techniques for rigorous, system-level spacecraft software requirements specification. While their work explicitly addresses behavior as well as function and performance, their focus is on techniques for generating component-based, reusable software specifications, rather than on developing a general approach for describing and reasoning about spacecraft system behavior. Furthermore, the SpecTRM-RL behavior modeling language used in both efforts, while formal, lacks the compositional and algebraic capabilities of CSP.

Stadter [97] discusses autonomous coordination of multiple spacecraft in a formation. Stadter and his colleagues make use of finite state automata to model spacecraft-spacecraft
interactions, and apply discrete event control theory to develop policies for coordinating these interactions. The emphasis in this case is more on synthesis of provably correct systems, rather than verification of existing designs. Like Hinchey’s work, Stadter focuses on interactions between spacecraft, rather than the behavior of an individual spacecraft.

2.4 Summary

The introduction of formal mathematical models into the design process for computer systems has produced tangible benefits in terms of both reductions in cost, and improvements in quality. As a result, there is an ongoing effort within the computer science community to bring greater formality to the design of computer software and hardware.

The CSP approach to formally modeling concurrent systems represents one instance of the increasing formalization of computer design. CSP has an active research community, with a good history of being able to move theoretical results into practice. The CSP approach has been successfully applied to a wide range of industrial problems, with extremely good results.

Applying some kind of formal approach to the specification and design of spacecraft behavior has obvious benefits. While there is some existing work on the formalization of certain aspects of spacecraft behavior, this work has focused primarily on the behavior spacecraft software, rather than considering the system-level behavior of the spacecraft as a whole.
Chapter 3
The Process Algebra CSP

“Mathematics itself provides an outstanding example of
the control of complexity by structure and abstraction.”
– C. A. R. Hoare

Communicating Sequential Processes (CSP) has its origins in a 1978 paper by Hoare [98]. That paper presented a simple pseudo-language for describing message-passing concurrent systems. In subsequent years Hoare, Brookes, and Roscoe worked to develop a mathematical theory suitable for reasoning about systems written in languages such as Hoare’s CSP, culminating in the publication of the paper “A Theory of Communicating Sequential Processes” [1] in 1984. The mathematical theory, originally referred to as Theoretical CSP, eventually displaced the original pseudo-language, and came to be known simply as CSP.

Specification and verification using CSP was originally based largely on the denotational proof methods outlined in Hoare’s 1985 book on the subject [2]. However, modern use of CSP for specification is closely associated with the FDR model-checker [48], a commercial tool that performs mechanical state-exploration of CSP processes in order to verify behavioral properties. The definitive work on modern CSP is Roscoe’s *The Theory and Practice of Concurrency* [3], which includes detailed coverage of both the theory of CSP, and of the techniques used by FDR to perform its verifications.

This chapter provides a brief introduction to the essentials of CSP notation and theory. This introduction should be sufficient to allow the casual reader to understand the rest of this dissertation. However, it is far from comprehensive. If a thorough understanding of CSP is desired, the interested reader is highly encouraged to seek out the texts on CSP by Roscoe [3] or Schneider [99].
3.1 CSP Notation

The behavior of a CSP process is completely defined by the pattern of interactions or communications that it has with its environment, i.e. other processes with which it is interacting. These interactions consist of discrete atomic events, which are assumed to occur only when all participants in the interaction are prepared to engage in the event. An intuitive way to think of a process is as a black box with various buttons or connectors which define its interface to the rest of the world (fig. 3.1). In this view of a process, the buttons or connectors represent the events that a process might engage in. The behavior of the process is then represented by the order in which buttons are pressed, or connectors generate signals.

Like all process algebras [4], CSP represents the behavior of a system as a process, which defines sequences of events or actions that the system can be observed to perform. CSP process expressions can be manipulated through the algebraic laws defined as part of the CSP theory, and can be composed to form new process expressions using a variety of different concurrency and choice operators. CSP process descriptions are written in a mathematical notation which uses a variety of special symbols to represent the process algebraic operators. With the advent of automated tools for process verification, a machine-readable form of CSP based on the standard ASCII character set has also been developed. The discussion that follows will focus on presenting the standard mathematical notation for CSP. The reader is encouraged to consult the notation guide at the front of this dissertation for the corresponding machine-readable CSP symbols.

Fig. 3.1: Cartoon of a process as a black box with two events in its interface.


3.1.1 Events

CSP events are abstractions of real-world interactions or communications between systems. Events may be atomic names, such as up, or compound objects formed by combining two or more objects with an infix dot, such as c.0. If c is thought of as a “channel” which communicates objects of type T, then the declaration channel c : T defines the set of compound events \{c\} = \{c.t | t \in T\}. The notation \{c\} denotes the set of all compound events with prefix c.

The alphabet of a process is the set of all events in which the process can engage. A process expression specifies the sequences in which the events in its alphabet can occur. The simplest CSP process is STOP, which never engages in any event, and is usually used to represent an error state. Successful termination is represented by the SKIP process, which engages in the special event ✓ (pronounced “tick”), and then does nothing else.

3.1.2 Prefixing and Recursion

The prefix operator combines an event and a process to produce a new process. The process a → P can engage in the event a, following which it behaves as the process P.

Example 3.1.2.1. The process

\[\text{DoSomething} = \text{something} \rightarrow \text{STOP}\]

can engage in the event something, following which it behaves as the process STOP.

The behavior of a parameterized process depends on the values assigned to its parameters. If the parameterized process P(x) is defined as P(x) = x → STOP, then P(a) can engage in the event a, following which it behaves as STOP.

In addition to defining processes in terms of other processes, processes can also be defined recursively, in terms of themselves.

Example 3.1.2.2. The process description

\[\text{Days} = \text{sunrise} \rightarrow \text{sunset} \rightarrow \text{Days}\]
defines a process which can engage in \textit{sunrise}, then \textit{sunset}, and then repeat its behavior, thus producing an infinite series of alternating \textit{sunrise} and \textit{sunset} events. This process can be represented graphically as the transition system shown in fig. 3.2.

3.1.3 Parallel Composition

One of the most fundamental operators in any process algebra is the parallel composition operator, which allows the concurrent execution of its component processes. CSP offers several parallel composition operators. The conceptually simplest form of parallel composition in CSP is \textit{interleaved parallel} composition. The \textit{interleaved parallel} composition $P || Q$ allows the processes $P$ and $Q$ to execute concurrently. The composite behavior is an interleaving of events from the behaviors of $P$ and $Q$. When an event in which both $P$ and $Q$ can engage is to be performed, one of the processes is nondeterministically selected to engage in that event.

\textbf{Example 3.1.3.1.} Consider the two processes \textit{Transmitter} and \textit{Receiver}, and their interleaved composition \textit{CommSystem}:

\begin{align*}
\text{Transmitter} &= \text{transmit} \rightarrow \text{tx\_ended} \rightarrow \text{Transmitter} \\
\text{Receiver} &= \text{receive} \rightarrow \text{rx\_ended} \rightarrow \text{Receiver} \\
\text{CommSystem} &= \text{Transmitter} || \text{Receiver}
\end{align*}

The behavior of \textit{CommSystem} consists of the events associated with each component process interleaved in an arbitrary order, as the transition system shown in fig. 3.3 illustrates.

\begin{figure}[h]
\centering
\includegraphics[width=0.3\textwidth]{fig32}
\caption{Transition system for the \textit{Days} process.}
\end{figure}
A more complex form of parallel composition involves defining interactions or communications between the processes being composed. In this form of composition the processes operate under the constraint that they can only engage in certain specified events when both processes in the composition are simultaneously prepared to engage in those events.

The *interface parallel* composition $P \parallel [X] Q$ interleaves the behaviors of $P$ and $Q$, but requires the simultaneous participation of both processes to perform any event in the interface set, $X$. When $P$ and $Q$ simultaneously engage in an event from $X$, the processes are said to have *synchronized* on that event. Any events outside of the interface set can occur whenever one of the processes is prepared to engage in them.

The *alphabetized parallel* composition $P \parallel [A|B] Q$ requires both $P$ and $Q$ to synchronize on any event in the set $A \cap B$. Events outside of the set $A \cap B$ are interleaved, but constrained such that $P$ and $Q$ can only engage in events in their respective interface sets, $A$ and $B$. The *generalized* alphabetized parallel composition $\parallel i : I \cdot A(i) \circ P(i)$ constrains each $P(i)$ to only perform events in the corresponding interface set, $A(i)$, and requires all processes that have events in common to synchronize on those events.

![Transition system for the CommSystem process.](image-url)
Example 3.1.3.2. Consider the system

\[Thruster = on \rightarrow \text{thrust} \rightarrow off \rightarrow \text{SKIP}\]

\[Controller = \text{command} \rightarrow on \rightarrow off \rightarrow \text{SKIP}\]

\[ControlledThruster = Controller \parallel \{\{on, off\}\} \parallel Thruster\]

The events on and off can only occur when both processes are prepared to engage in them, while command can occur whenever Controller is ready to engage in it, and thrust is similarly dependent only on Thruster (see fig. 3.4).

Example 3.1.3.3. A deadlock situation could arise if, for example, command was added to the synchronization set of the ControlledThruster process defined in example 3.1.3.2. In that case, Controller could not engage in command until Thruster was also prepared to engage in that event, but Thruster would be similarly constrained awaiting a synchronization of the on event. As a result the composite process would not engage in any events, and thus

\[DeadlockedControlledThruster = Controller \parallel \{\{on, off, command\}\} \parallel Thruster\]

\[\equiv \text{STOP}\]

Fig. 3.4: The parallel composition ControlledThruster.
3.1.4 Alternative Composition

Another of the fundamental operators in process algebra is the alternative composition operator, which provides an alternative, or choice, between two different processes. CSP actually incorporates two different choice operators, with slightly different properties.

The external choice operator provides a choice between two alternative behaviors. The process $P \parallel Q$ can behave as either $P$ or $Q$. If the environment initially offers an event in which only $P$ is willing to engage, the process behaves as $P$. The offer of an event in which only $Q$ is willing to engage causes the process to behave as $Q$. The offer of events in which both $P$ and $Q$ are willing to engage results in a nondeterministic resolution of the choice.

The generalized external choice operator provides the environment with a choice between several alternative behaviors. The process $\parallel i : I \cdot P(i)$ can behave as any of the processes $P(i)$, for $i \in I$.

Example 3.1.4.1. The composite process

\[
\text{Transceiver} = \text{Transmitter} \parallel \text{Receiver}
\]

can behave either like the process Transmitter, or like the process Receiver (defined in sec. 3.1.3). The choice of which behavior Transceiver will exhibit is determined by the first event that the environment elects to engage in: if the environment engaged in transmit, the process Transceiver would also engage in transmit, and then continue to behave like Transmitter.

Intuitively speaking, the composite Transceiver process is like a box with two buttons on it: pressing the transmit button activates the transmitter, and pressing the receive button activates the receiver (fig. 3.5).

The nondeterministic choice operator, also known as “internal choice,” does not allow the environment to exercise any control over the resolution of the choice. The process $P \cap Q$ can elect to behave as either $P$ or $Q$. It can offer to engage in the initial events of either $P$ or $Q$, but is not required to engage in either. If the environment attempts to engage in an event in which the process refuses to engage, the result may be deadlock. However, if the
environment offers events from both $P$ and $Q$, the process must respond by continuing to behave as one of $P$ or $Q$.

**Example 3.1.4.2.** The process

\[
Radio = Transmitter \sqcap Receiver
\]

may behave either like the process $Transmitter$, or the process $Receiver$, but there is no way to select which behavior $Radio$ will exhibit.

In terms of the intuitive “box with buttons” representation, the $Radio$ process is like a box with faulty buttons: pressing just one button might or might not activate the radio; pressing both buttons together will activate the radio, but whether it transmits or receives once activated cannot be controlled. Nondeterministic choice is primarily used to abstract away the internal details of how a process makes decisions about what behavior to exhibit. It is also often used in writing specifications, when, for example it is sufficient that $Radio$ behave as one of $Transmitter$ or $Receiver$, but does not necessarily have to behave as both.

### 3.1.5 Channels

Communications over a channel $c : T$ can be described using the notations $c?x$ and $c!x$, denoting input and output respectively. The process $P = c?x \rightarrow c!x \rightarrow P$ is initially willing to engage in any event from the set $\{c.t \mid t \in T\}$, after which the variable $x$ takes on a corresponding value $t \in T$. Subsequent engagement in the output event $c!x$ is then
equivalent to a second occurrence of $c.t$, after which the process is again ready to engage in another input event. The parallel composition $P \| \{c\} \parallel Q$ requires $P$ and $Q$ to synchronize on all events in $\{c\}$, allowing $Q$ to first send a value $t$ to $P$ through $c$, and then to receive that value back again.

**Example 3.1.5.1.** If we define the set $BITS = \{0, 1\}$, then the process

\[
\text{channel } left, right : BITS
\]

\[
CopyBits = (left.0 \to right.0 \to CopyBits)
\]

\[
\quad \square (left.1 \to right.1 \to CopyBits)
\]

can receive members of the set $BITS$ along the $left$ channel, and will output each received “bit” along the $right$ channel (see fig. 3.6).

The $CopyBits$ process can be written more compactly by using the symbols $?$ and $!$ to denote input and output respectively:

\[
CopyBits = left?x : BITS \to right!x \to CopyBits
\]

Fig. 3.6: Transition system for the $CopyBits$ process.
Example 3.1.5.2. We define a process $\text{SendBits}$, which sends an alternating string of 0 and 1 bits, and place it in parallel with $\text{CopyBits}$:

$$
\text{SendBits} = \text{left}!0 \rightarrow \text{left}!1 \rightarrow \text{SendBits} \text{MoreBits} = \text{SendBits} \parallel \left\{ \text{left} \right\} \parallel \text{CopyBits}
$$

where the notation $\left\{ \text{left} \right\}$ represents the set of all compound events that start with $\text{left}$, i.e. $\left\{ \text{left} \right\} = \{ \text{left}.0, \text{left}.1 \}$. Since $\text{SendBits}$ and $\text{CopyBits}$ synchronize on any events involving channel $\text{left}$, the result is that $\text{CopyBits}$ effectively “receives” bit values over this channel from $\text{SendBits}$ (and then promptly outputs those bit values on channel $\text{right}$). Figure 3.7 shows the transition system created by the parallel composition.

Figure 3.8 shows the process structure of the $\text{MoreBits}$ process. As the diagram indicates, using channel events for communication does not prevent other processes from also synchronizing on events involving those channels. If additional synchronization is not desirable it can be prevented using the hiding operator, which will be introduced in sec. 3.1.7.

3.1.6 Sequential Composition

The sequential composition operator is used in conjunction with the $\text{SKIP}$ process to combine two processes such that the composite behavior is that of the first process followed by that of the second process. The process $P ; Q$ behaves as process $P$ until $P$ internally reaches a $\text{SKIP}$, after which the process behaves as $Q$. The generalized sequential composition operator allows a list of processes to be executed in sequence. The process $P = ; p : Ps \bullet p$ behaves in turn as each of the processes $p$ in the list $Ps$. 

![Fig. 3.7: Transition system for the MoreBits process.](image)
Example 3.1.6.1. The process

\[ \text{Mission} = \text{Launch} \; ; \; \text{Operations} \; ; \; \text{Disposal} \]

exhibits the behavior of the processes \textit{Launch}, \textit{Operations}, and \textit{Disposal}, in that order. The transition from one process to the next occurs when a process internally reaches a \textit{SKIP}, and successfully terminates. If a process does not terminate, its successor is never activated. Thus if \textit{Operations} never terminates, then the behavior of \textit{Disposal} will never occur.

3.1.7 Hiding

It is sometimes useful to be able to abstract away events which are not directly relevant to an analysis, such as concealing “internal” communications in a parallel composition. This type of abstraction can be performed using the hiding operator. As an example, recall the \textit{Morebits} process from the discussion of channels. Using the hiding operator we can write

\[ \text{MoreBits}_2 = \text{MoreBits} \setminus \{\text{left}\} \]

This conceals all events involving the \textit{left} channel, thereby making the observable behavior of \textit{MoreBits}_2 simply an alternating string of \textit{right}.0 and \textit{right}.1 events. Figure 3.9 illustrates the resulting composite process.

The use of hiding requires some care, since it may change a previously deterministic process into one that is nondeterministic, as example 3.1.7.1 demonstrates. Hiding also has the potential to the introduce \textit{divergence}, as described in example 3.1.7.2.
Example 3.1.7.1. If we rewrite the Transceiver process from the discussion of alternative composition as

\[ Transceiver_2 = (Transmitter \Box Receiver) \setminus \{transmit, receive\} \]

then the environment can no longer exercise control over which behavior \( Transceiver_2 \) will exhibit, that of \( Transmitter \) or of \( Receiver \).

Example 3.1.7.2. If all events in the recursive \( Days \) process are hidden

\[ Days = (sunrise \rightarrow sunset \rightarrow Days) \setminus \{sunrise, sunset\} \]

then we are left with a process which never produces an externally observable event, but also never terminates. Note that the special termination event \( \check{\surd} \) can never be hidden.

A process such as the one above is referred to as a divergent process. Divergence is represented by the primitive process \( Div \), which does nothing but diverge.

3.1.8 Scoping

The scoping construct \texttt{let s = expression within P} causes the symbol \( s \) to take on the value of \( expression \), but only within the scope of \( P \). This permits the symbol \( s \) to be used to simultaneously denote different expressions in different contexts. The scoping construct also provides a convenient way to group several related processes.
Example 3.1.8.1. The CommSystem process of example 3.1.3.1 can be rewritten such that
the Transmitter and Receiver processes are only defined within the scope of CommSystem.

\[
\text{CommSystem} = \\
\text{let} \\
\text{Transmitter} = \text{transmit} \rightarrow \text{tx\_ended} \rightarrow \text{Transmitter} \\
\text{Receiver} = \text{receive} \rightarrow \text{rx\_ended} \rightarrow \text{Receiver} \\
\text{within} \\
\text{Transmitter} \parallel \text{Receiver}
\]

3.2 CSP Theory

In addition to being a well-defined language for precisely describing concurrent systems,
CSP is also a theory of concurrency, and as such provides a rich set of theoretical tools for
understanding and manipulating systems described using CSP expressions.

The theoretical foundations of CSP can, and have been, expressed in three different
forms: denotational models, algebraic laws, and operational semantics. Each of these ways
of expressing the “meaning” of a CSP process is mutually consistent with the others, and
each is useful in different ways. All three of the theoretical approaches will be introduced
in the discussion that follows.

3.2.1 Denotational Models

Although CSP is a process algebra, much of the theoretical work in CSP has tradi-
tionally been carried out using various denotational models. A denotational model is a
mathematical construct that can be used to provide an abstract view of processes. This
abstract view permits more general reasoning about classes of processes, and allows us to
make precise statements about what it means for two processes to be “equivalent.” The
mapping from algebraic notation to denotational model is carried out through a collection of
translation rules known as a denotational semantics. In CSP, the denotational models have
typically been based on sets and sequences of events. The three major CSP denotational
models are the\textit{traces} model, the\textit{stable failures} model, and the\textit{failures/divergences} model.

\textbf{Traces}

The\textit{traces} model is the most basic of the denotational models. A \textit{trace} is simply a
sequence of events. The simplest way to describe the observed behavior of a process is as a
set of traces which record all of the possible sequences of interactions between the process
and its environment. Traces are essentially another name for the \textit{strings} used in automata
theory [100], and a trace set is equivalent to the \textit{language} of an automaton.

\textbf{Example 3.2.1.1.}

\[
\text{traces}(STOP) = \{\langle \rangle\}
\]
\[
\text{traces}(a \rightarrow b \rightarrow STOP) = \{\langle \rangle, \langle a \rangle, \langle a, b \rangle\}
\]
since \textit{STOP} never engages in any events, while the process \((a \rightarrow b \rightarrow STOP)\) can be
observed to have done nothing, to have engaged in event \textit{a}, or to have engaged in event \textit{a}
and then event \textit{b}.

Traces are defined as sequences drawn from the set \(\Sigma^*\checkmark\), where

- \(\Sigma\) is the set of all possible events (or “universal alphabet”) for the system under
consideration.

- \(\Sigma^*\) is the set of all finite sequences (including the empty sequence) that can be formed
from elements of \(\Sigma\).

- \(\Sigma^*\checkmark = \Sigma^* \cup \{s \checkmark \mid s \in \Sigma^*\}\), i.e. the termination event \(\checkmark\) is always the last event
in a trace.
Definition 1 (see Roscoe [3]). A process in the traces model is defined as a subset \( traces(P) \subseteq \Sigma^* \) satisfying the following conditions:

\[
\begin{align*}
T1. \quad & \langle \rangle \in traces(P) \quad (3.1) \\
T2. \quad & s_1 \triangleright s_2 \in traces(P) \implies s_1 \in traces(P) \quad (3.2)
\end{align*}
\]

That is, \( traces(P) \) always contains at least the empty trace (\( \langle \rangle \)), and \( traces(P) \) is prefix closed (i.e. if a trace is a member of \( traces(P) \) then the shorter traces representing earlier observations of \( P \) must also be in \( traces(P) \)).

The traces of a composite process can be determined from the traces of its component processes using the rules of the denotational semantics. For example, the rule for composition using the external choice operator is

\[
traces(P \boxdot Q) = traces(P) \cup traces(Q)
\]

where \( P \) and \( Q \) are arbitrary processes. The complete denotational semantics for the traces model (see Roscoe’s text [3]) provides rules for all of the CSP operators, and thus permits any process description to be mapped to a set of traces. If \( traces(P) = traces(Q) \) then the two processes \( P \) and \( Q \) are said to be trace equivalent.

The traces model is a good way of describing the behavior that a process may engage in. For some purposes, such as ensuring that certain events can never occur, this model is sufficient. However, the traces model is not able to describe the behaviors that a process must engage in. As a result, some types of processes cannot be differentiated from each other in the traces model.

Example 3.2.1.2.

\[
\begin{align*}
traces((a \rightarrow STOP) \boxdot (b \rightarrow STOP)) &= \{\langle \rangle, \langle a \rangle, \langle b \rangle\} \\
traces((a \rightarrow STOP) \sqcap (b \rightarrow STOP)) &= \{\langle \rangle, \langle a \rangle, \langle b \rangle\}
\end{align*}
\]
The traces of the two processes are the same, since both processes may engage in events a or b. However, the external choice process must engage in an a or b event if the environment attempts to communicate such an event. The nondeterministic choice makes no such guarantee.

If we wish to be able to make a distinction between the two processes above, we need to add more information to the traces model. This is the rationale for the introduction of the stable failures model.

**Stable Failures**

The stable failures model extends the traces model with the idea of refusal sets. A refusal is a set of events $X \subseteq \Sigma^\vee$ (where $\Sigma^\vee = \Sigma \cup \{\checkmark\}$) which a process may choose to refuse to engage in. The idea of using "refusals" instead of "acceptances" (i.e. those events a process is prepared to engage in) can be a little counter-intuitive at first. However, refusal sets have historically been the standard way of describing process behavior in CSP. Refusal sets are apparently preferred over acceptance sets because refusals permit a simpler approach at the theoretical level [1], although some authors have used acceptance-based semantics to their advantage [101].

A *failure* is a pair $(s, X)$, consisting of a trace $s$, and a refusal $X$ which identifies the events in which a process may refuse to engage once it has executed the trace $s$. The observed behavior of a process in the stable failures model is described by the pair $(\text{traces}(P), \text{failures}(P))$. Assuming $\Sigma = \{a, b\}$, then for the processes from example 3.2.1.2:

\[
\text{failures}((a \rightarrow \text{STOP}) \square (b \rightarrow \text{STOP})) = \{(\emptyset, \emptyset), (\langle a \rangle, \{a, b\}), (\langle b \rangle, \{a, b\})\}
\]

\[
\text{failures}((a \rightarrow \text{STOP}) \sqcap (b \rightarrow \text{STOP})) = \{(\emptyset, \{a\}), (\emptyset, \{b\}), (\langle a \rangle, \{a, b\}), (\langle b \rangle, \{a, b\})\}
\]

The failures of the two processes are clearly different, since the external choice cannot initially refuse $a$ or $b$, while the nondeterministic choice can initially refuse $a$ or $b$, but not both. Thus, while these two processes are trace equivalent, they are not failures equivalent.
**Definition 2** (see Roscoe [3]). A process in the stable failures model is a pair

$$(\text{traces}(P), \text{failures}(P)),$$

where $\text{traces}(P)$ follows definition 1, and $\text{failures}(P) \subseteq \Sigma^* \times \mathcal{P}{\Sigma^*}$.

The pair $(\text{traces}(P), \text{failures}(P))$ must satisfy the conditions:

1. $\langle \rangle \in \text{traces}(P)$, \hspace{1cm} (3.3)
2. $s_1 \mathbin{<} s_2 \in \text{traces}(P) \implies s_1 \in \text{traces}(P)$ \hspace{1cm} (3.4)
3. $(s, X) \in \text{failures}(P) \implies s \in \text{traces}(P)$ \hspace{1cm} (3.5)
4. $(s, X) \in \text{failures}(P) \land Y \subseteq X \implies (s, Y) \in \text{failures}(P)$ \hspace{1cm} (3.6)
5. $(s, X) \in \text{failures}(P) \land \forall a \in Y \cdot s \mathbin{<} \langle a \rangle \notin \text{traces}(P)$ \hspace{1cm} (3.7)
6. $s \mathbin{<} \langle \checkmark \rangle \in \text{traces}(P) \implies (s, \Sigma) \in \text{failures}(P)$ \hspace{1cm} (3.8)
7. $s \mathbin{<} \langle \checkmark \rangle \in \text{traces}(P) \implies (s \mathbin{<} \langle \checkmark \rangle, X) \in \text{failures}(P)$ \hspace{1cm} (3.9)

The conditions on the $(\text{traces}(P), \text{failures}(P))$ pair essentially state that:

- The traces portion of the pair must be a valid trace by the conditions of the trace model (conditions T1 and T2).
- The trace associated with any failure must appear in the trace set (condition SF1).
- A process can refuse any subset of a given refusal set (condition SF2).
- The failures set for a given state must include any events that the process can never perform when in that state (condition SF3).
- A process able to terminate can refuse to do anything else (condition SF4).
- The behavior of all processes after termination looks the same (condition SF5).
As with the traces model, the denotational semantics associated with the failures model provides rules for determining the failure set of a composite process from the failure sets of its components. Simple examples of these rules, which would be sufficient to develop some of the failure sets presented above, include

\[
\text{failures}(\text{STOP}) = \{ (\langle \rangle, X) \mid X \subseteq (\Sigma \cup \checkmark) \} \\
\text{failures}(a \to P) = \{ (\langle \rangle, X) \mid a \notin X \} \cup \{ (\langle a \rangle \triangleleft s, X) \mid (s, X) \in \text{failures}(P) \} \\
\text{failures}(P \sqcap Q) = \text{failures}(P) \cup \text{failures}(Q)
\]

The full denotational semantics for the stable failures model can be found in Roscoe [3].

**Failures/Divergences**

The final model that we will discuss is the failures/divergences model. This model essentially extends the failures model to handle the concept of *divergence*. Recall that divergence was previously mentioned in sec. 3.1.7, where it was informally described as being the behavior of a process which stops producing externally observable events, and does not terminate. In the example discussed in sec. 3.1.7, divergence was created by hiding all of the observable events of the example process. In terms of the theory of CSP, the hiding operation translates an observable event into the unobservable internal event $\tau$. A divergent process is then formally defined as a process which, while it can exhibit observable behavior, can also produce an unbroken infinite string of $\tau$ events. So hiding all of the observable events in an infinite process will necessarily result in a process which is divergent. Describing divergence provides one more means of differentiating between two different processes that would otherwise be indistinguishable.

**Definition 3** (see Roscoe [3]). A process in the failures/divergences model is a pair

\[
(\text{failures}_\perp(P), \text{divergences}(P))
\]
where $divergences(P)$ is defined as the set of all traces that can lead to divergent behavior (i.e. an unbroken infinite string of $\tau$ events) on the part of $P$, and $failures_{\perp}(P) = failures(P) \cup \{ (s, X) \mid s \in divergences(P) \}$. Thus, $failures_{\perp}(P) \subseteq \Sigma^* \times \mathcal{P}\Sigma^\triangledown$ and $divergences(P) \subseteq \Sigma^*\triangledown$. The pair $(failures_{\perp}(P), divergences(P))$ must satisfy the conditions:

\[
\begin{align*}
F1. & \quad \langle \rangle \in \{ s \mid (s, X) \in failures_{\perp}(P) \} & (3.10) \\
F2. & \quad s_1 \triangledown s_2 \in \{ s \mid (s, X) \in failures_{\perp}(P) \} \implies s_1 \in \{ s \mid (s, X) \in failures_{\perp}(P) \} & (3.11) \\
F3. & \quad (s, X) \in failures_{\perp}(P) \land Y \subseteq X \implies (s, Y) \in failures_{\perp}(P) & (3.12) \\
F4. & \quad (s, X) \in failures_{\perp}(P) \land \forall a \in Y \cdot s \triangledown \langle a \rangle \notin \{ s \mid (s, X) \in failures_{\perp}(P) \} \\
& \quad \implies (s, X \cup Y) \in failures_{\perp}(P) & (3.13) \\
F5. & \quad s \triangledown \langle \triangledown \rangle \in \{ s \mid (s, X) \in failures_{\perp}(P) \} \implies (s, \Sigma) \in failures_{\perp}(P) & (3.14) \\
D1. & \quad s_1 \in divergences(P) \cap \Sigma^* \land s_2 \in \Sigma^\triangledown \implies s_1 \triangledown s_2 \in divergences(P) & (3.15) \\
D2. & \quad s \in divergences(P) \implies (s, X) \in failures_{\perp}(P) & (3.16) \\
D3. & \quad s \triangledown \langle \triangledown \rangle \in divergences(P) \implies s \in divergences(P) & (3.17)
\end{align*}
\]

Conditions F1 through F5 impose essentially the same constraints as the corresponding conditions in the stable failures model, although here expressed in terms of the implicit traces contained in $failures_{\perp}(P)$. Condition D1 states that we will not bother to distinguish the behavior of processes once they have diverged. Condition D2 ensures correspondence between $divergences(P)$ and $failures_{\perp}(P)$. Condition D3 ensures that we ignore the behavior of all processes after successful termination.

It is interesting to note that the failures/divergences model does not explicitly include $traces(P)$ in its process description, but instead includes the traces implicitly in $failures_{\perp}(P)$. This was not possible in the stable failures model, because $failures(P)$ does not include any traces which result in divergence.

The complete denotational semantics for the failures/divergences model can be found in Roscoe’s text [3], and, much like the denotational semantics for the other models, provides
3.2.2 Algebraic Laws

Since CSP is a process algebra, it naturally includes a set of algebraic laws. The algebraic laws of CSP provide a way to reconfigure and simplify process descriptions through direct manipulation of the symbols of the process description. The laws themselves are intimately tied to the denotational models discussed in the previous section: an algebraic equivalence between two processes implies that the processes are also equivalent in terms of a given denotational model. Since each of the denotational models captures slightly different process equivalences it is sometimes necessary to specify the type of equivalence (traces, failures, or failures/divergences) for which a particular algebraic law is valid. However, the majority of the algebraic laws generalize across all of the denotational models, and thus provide a powerful tool for process manipulation. Some of the key algebraic laws of CSP are presented below. In presenting these laws we will follow the naming scheme used by Roscoe [3], in which each law is labeled to indicate both the operator(s) to which it applies, and the property it represents.

There are certain fundamental algebraic laws that are shared by all process algebras [4]. The CSP versions of these laws [3] appear below.

- Commutativity (symmetry), associativity, and idempotency of choice composition

\[
P \parallel Q = Q \parallel P \quad \langle \parallel\text{-sym} \rangle
\]
\[
P \sqcap Q = Q \sqcap P \quad \langle \sqcap\text{-sym} \rangle
\]
\[
P \parallel (Q \parallel R) = (P \parallel Q) \parallel R \quad \langle \parallel\text{-assoc} \rangle
\]
\[
P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap R \quad \langle \sqcap\text{-assoc} \rangle
\]
\[
P \parallel P = P \quad \langle \parallel\text{-idem} \rangle
\]
\[
P \sqcap P = P \quad \langle \sqcap\text{-idem} \rangle
\]
• Associativity and distributivity of sequential composition

\[(P ; Q) ; R = P ; (Q ; R) \quad \langle ; -assoc \rangle\]

\[(P \sqcap Q) ; R = (P ; R) \sqcap (Q ; R) \quad \langle ; -dist-l \rangle\]

\[P ; (Q \sqcap R) = (P ; Q) \sqcap (P ; R) \quad \langle ; -dist-r \rangle\]

• Commutativity and associativity of parallel composition

\[P \parallel [X] Q = Q \parallel [X] P \quad \langle [X]-sym \rangle\]

\[P \parallel Q = Q \parallel P \quad \langle ||-sym \rangle\]

\[(P \parallel [X] Q) \parallel [X] R = P \parallel [X] (Q \parallel [X] R) \quad \langle [X]-assoc \rangle\]

\[(P \parallel Q) \parallel R = P \parallel (Q \parallel R) \quad \langle ||-assoc \rangle\]

As a result of the pathological case in which one of the processes in a \(\square\) composition is \(\text{SKIP}\), external choice is not distributive over \(;\). Since the environment cannot exercise control over the \(\checkmark\) event, \((\text{SKIP} \square Q) ; R = R \sqcap (R \square (Q ; R))\) [1].

In addition to the preceding fundamental process algebraic laws, CSP possesses a variety of other laws involving the distributivity of its operators, including [3]:

\[P \square (Q \sqcap R) = (P \square Q) \sqcap (P \square R) \quad \langle \square \text{-dist} \rangle\]

\[P \sqcap (Q \square R) = (P \sqcap Q) \sqcap (P \square R) \quad \langle \sqcap \square \text{-dist} \rangle\]

\[a \rightarrow (P \sqcap Q) = (a \rightarrow P) \sqcap (a \rightarrow Q) \quad \langle \text{prefix-dist} \rangle\]

\[P \parallel [X] (Q \sqcap R) = (P \parallel [X] Q) \sqcap (P \parallel [X] R) \quad \langle [X]\text{-dist} \rangle\]

\[P \parallel (Q \sqcap R) = (P \parallel Q) \sqcap (P \parallel R) \quad \langle ||\text{-dist} \rangle\]

Another important class of laws is the so-called “step laws,” which define how a process evolves in response to a single event. These laws are essentially operational in nature, in that they define transitions of a process state to another state, and are closely related to
the operational semantics described in the next section. Step laws for each CSP operator can be found in Roscoe [3]. An example of a step law is the external-choice step law:

\[
(\Box x : A \cdot P) \Box (\Box x : B \cdot Q) = \Box x : A \cup B \cdot \begin{cases} P & x \notin B \\ Q & x \notin A \end{cases} \quad \langle \Box\text{-step} \rangle
\]

Example 3.2.2.1. Consider the process

\[
Sys = ((P \Box P) [X] ((b \rightarrow (Q \cap R)) \Box (a \rightarrow Q))) \\
\cap (P [X] ((a \rightarrow R) \Box (b \rightarrow (R \cap Q))))
\]

Through successive application of several of the algebraic laws defined above we can rearrange the complicated \(Sys\) process description into something far less complex:

\[
Sys = (P [X] ((b \rightarrow (Q \cap R)) \Box (a \rightarrow Q))) \\
\cap (P [X] ((a \rightarrow R) \Box (b \rightarrow (Q \cap R)))) \quad \text{by } \langle \Box\text{-idem} \rangle
\]

= \(P [X] ((b \rightarrow (Q \cap R)) \Box (a \rightarrow Q))) \quad \text{by } \langle \cap\text{-sym} \rangle
\]

= \(P [X] ((a \rightarrow R) \Box (b \rightarrow (Q \cap R)))) \quad \text{by } \langle \cap\text{-sym} \rangle
\]

= \(P [X] ((a \rightarrow R) \Box (a \rightarrow Q))) \quad \text{by } \langle \Box\text{-sym} \rangle
\]

= \(P [X] ((b \rightarrow (Q \cap R)) \Box (a \rightarrow Q)) \quad \text{by } \langle \Box\text{-dist} \rangle
\]

= \(P [X] ((b \rightarrow (Q \cap R)) \Box ((a \rightarrow Q) \cap (a \rightarrow R))) \quad \text{by } \langle \Box\text{-dist} \rangle
\]

= \(P [X] ((b \rightarrow (Q \cap R)) \Box (a \rightarrow (Q \cap R))) \quad \text{by } \langle \text{prefix-dist} \rangle
\]

= \(P [X] (\Box x : \{a, b\} \cdot (Q \cap R)) \quad \text{by } \langle \Box\text{-step} \rangle
\]
The process description which results from the application of the algebraic laws is behaviorally equivalent to the original process, but is clearly much easier to read and understand. More importantly, if the original process description represented a system design created by composing several existing modules, then the simplified process description shows how to achieve the same behavior with a significantly less complex design. The ability to simplify process descriptions in this way has obvious practical benefits in terms of the cost and reliability of implementing desired behaviors.

3.2.3 Operational Semantics

An operational semantics is a way to describe the “meaning” of a program or process in terms of transitions from one process state to another one. In the context of CSP, transitions are equivalent to events, and process states represent the events that a process is ready to accept or refuse once it has completed a given transition. The operational semantics for CSP provides a set of rules (so-called “firing rules”) which define how the state of a process, or a composition of processes, will evolve in response to an event.

The rules of the CSP operational semantics provide a convenient way to rigorously map a process description to a labeled transition system (LTS). This is useful for two reasons: firstly, an LTS can easily be converted into a visual representation of the system (such as the illustrative transition systems that we presented in sec. 3.1), which can sometimes aid understanding of system behavior; secondly, an LTS provides a complete representation of the “state space” of a process, which can then be exhaustively explored in order to verify that some desired set of properties holds in every possible state. The industrial CSP verification tool FDR makes use of LTS state exploration in its verification checks.

The operational semantics for CSP are traditionally expressed in the form of inference rules. A simple example of these inference rules is the following rule set, which provides the operational semantics for the external choice operator:
These rules essentially state that if a process $P$ can undergo a transition $a$ (i.e. engage in the event $a$) to a state $P'$ then such a transition resolves the external choice, and that internal transitions ($\tau$) do not resolve the external choice.

A similar set of rules exist for the interface parallel operator (and can be generalized to apply to both interleaving and alphabetized parallel):

$$
\begin{align*}
\frac{P \xrightarrow{a} P'}{P \parallel Q \xrightarrow{a} P'} & \quad \frac{P \xrightarrow{\tau} P'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q} \\
\frac{Q \parallel P \xrightarrow{a} P'}{Q \parallel P \xrightarrow{\tau} Q \parallel P'} & \quad \frac{Q \parallel P \xrightarrow{\tau} Q \parallel P'}
\end{align*}
$$

In this case, the inference rules state that any event which is in the synchronization set $X$ will cause both $P$ and $Q$ to change state, while events outside of $X$ (including $\tau$ events) cause only a single process to change state.

A full set of inference rules for all of the CSP operators can be found in both Roscoe’s [3] and Schneider’s [99] texts. However, even the two simple inference rules we have so far are sufficient to examine a small example of converting a process description into an LTS.

**Example 3.2.3.1.** Consider the system

$$
P = (a \rightarrow P) \parallel (b \rightarrow c \rightarrow P)
Q = b \rightarrow c \rightarrow (Q \parallel (d \rightarrow Q))
P \parallel \{b, c\} \parallel Q
$$

From the external choice inference rules we see that $P$ will initially either engage in $a$ and then return to its initial state, or engage in $b$ and enter a state in which the only event it can perform is $c$. From the parallel inference rules we see that $Q$ can initially only engage
in $b$, and therefore must synchronize with $P$ to make any progress. Thus, we obtain

$$(P \parallel \{b, c\} \parallel Q) \xrightarrow{a} (P \parallel \{b, c\} \parallel Q)$$
$$(P \parallel \{b, c\} \parallel Q) \xrightarrow{b} ((c \rightarrow P) \parallel \{b, c\} \parallel (c \rightarrow (Q \sqcap (d \rightarrow Q))))$$

Once $P$ and $Q$ synchronize on $b$ the only possible event that can occur next in both processes is $c$. The occurrence of $c$ will return $P$ to its initial state (i.e. able to engage in $a$ or $b$), and cause $Q$ to reach a state in which it has an external choice between its initial state (i.e. ready to engage in $b$) or engaging in $d$ and then returning to its initial state.

$$( (c \rightarrow P) \parallel \{b, c\} \parallel (c \rightarrow (Q \sqcap (d \rightarrow Q)))) \xrightarrow{c} (P \parallel \{b, c\} \parallel (Q \sqcap (d \rightarrow Q)))$$
$$(P \parallel \{b, c\} \parallel (Q \sqcap (d \rightarrow Q))) \xrightarrow{d} (P \parallel \{b, c\} \parallel Q)$$

Assembling each of these system states, and the possible transitions in each state, into an LTS, we obtain the transition system depicted in fig. 3.10.

### 3.3 Specification and Verification with CSP

CSP is more than just a mathematical notation for describing and manipulating concurrent systems. It is also a useful tool for specifying the properties that the behavior of a system should exhibit, and for verifying that a design actually meets its specification. There are two standard approaches to specifying and verifying behavior in CSP. One approach involves specification of predicates on a denotational model, followed by verification
of predicate satisfaction through logical deduction. The other approach expresses both specification and implementation as processes, and provides verification by checking for a relationship between the two processes known as refinement. Researchers working on other process algebras are developing a third approach [102, 103], based on algebraic verification of process properties, but this approach is still in its infancy, and has not yet seen much application in the CSP community.

3.3.1 Predicate Satisfaction

The predicate satisfaction approach to process specification was introduced in Hoare’s original text on CSP. Specifications are expressed as predicates on the traces or failures of a process. The predicates are written as sat clauses such as $P \text{ sat} \ Spec(s, X)$, where

$$P \text{ sat} \ Spec(s, X) \iff \forall (s, X) \in \text{failures}(P) \bullet Spec(s, X)$$

Example 3.3.1.1. The sat clause

$$CopyBits \text{ sat} \ (s \downarrow right \leq s \downarrow left)$$

states that in any trace $s$ of the CopyBits process the sequence of values produced on channel right is always a prefix of the sequence of values it has received on channel left. ◻

Hoare’s text provides a number of deductive rules that permit a proof of predicate satisfaction for a composite process to be built up from the sat clauses of the component processes. The sat-based approach specification and verification has two advantages: it allows for a natural expression of desired behavioral properties, and it avoids explication of the state space, making large or infinite state systems tractable. The drawback is that verification must be performed by manual proof (perhaps aided by a proof assistant such as Isabelle/HOL or PVS), although Martin [65] has recently introduced a tool capable of performing automatic sat checking for a limited subset of specifications.
3.3.2 Process Refinement

The refinement approach to specification and verification considers both specification and implementation as processes. An implementation process is considered acceptable with respect to a specification if it exhibits some subset of the behavior of the specification, in which case it is said to “refine” the specification. Refinement must be evaluated with respect to one of the denotational models described in sec. 3.2.1.

Definition 4 (see Roscoe [3]). Refinement in the traces, stable failures, and failures/divergences models is defined as:

\[ \text{Spec} \sqsubseteq_T \text{Impl} \iff \text{traces(Impl)} \subseteq \text{traces(Spec)} \]

\[ \text{Spec} \sqsubseteq_F \text{Impl} \iff (\text{traces(Impl)} \subseteq \text{traces(Spec)}) \land \text{failures(Impl)} \subseteq \text{failures(Spec)}) \]

\[ \text{Spec} \sqsubseteq_{FD} \text{Impl} \iff (\text{failures}(\|	ext{Impl}) \subseteq \text{failures}(\|	ext{Spec}) \land \text{divergences(Impl)} \subseteq \text{divergences(Spec)}) \]

Example 3.3.2.1. Since \text{traces(Transmitter)} \subseteq \text{traces(Transmitter } \cap \text{ Receiver)}, we can state that

\[ \text{Transmitter } \cap \text{ Receiver} \sqsubseteq_T \text{ Transmitter} \]

The refinement relation is a partial order on the space of processes, and possesses several useful properties that permit systems to be designed and verified in a stepwise, compositional manner, including:

- Transitivity – \( P \sqsubseteq Q \land Q \sqsubseteq R \implies P \sqsubseteq R \).

- Monotonicity – \( P \sqsubseteq Q \implies C[P] \sqsubseteq C[Q] \), where \( C[\cdot] \) is any process algebraic expression with a free process identifier.
We are now in a position to understand the name of the FDR tool: FDR stands for *Failures/Divergences Refinement*. FDR is a tool that uses exhaustive state-space exploration of process descriptions to automatically test for traces, failures, or failures/divergences refinement of a specification by an implementation. The use of the transitivity and monotonicity properties of refinement, along with various behavior-preserving state-space compression techniques, allows FDR to be used to verify systems too complex to check by manual means. Although automated theorem-proving tools have been applied to CSP process verification, refinement checking with FDR appears to be the most commonly used method for specification and verification of processes in the CSP community, particularly in industrial applications. As a result, we will favor process-based specifications throughout the remainder of this dissertation.
Chapter 4

Specifying Spacecraft System Behavior

“When I use a word,” said Humpty Dumpty in rather a scornful tone, ‘it means just what I choose it to mean - neither more nor less.’”

– Lewis Carroll

The existing practice in most spacecraft design projects is to define the system-level behavior of a spacecraft in terms of loosely defined concepts such as “functions,” and “mode transitions,” and informal diagrams such as State Transition Diagrams and Functional Flow Block Diagrams. In this chapter we develop a CSP interpretation for some of the most common spacecraft behavior specification concepts. These CSP-based definitions demonstrate that process algebraic expressions can be used to describe spacecraft behavior, and helps to elucidate the relationships between different specification constructs. The resulting conceptual framework allows spacecraft behavior specification to be approached in a more structured and systematic way.

The chapter is structured as follows. Since the notion of “required functions” is a key part of most discussions of spacecraft requirements, we begin by establishing a precise definition of the term required function. Based on this definition, we draw a distinction between required functions and required behaviors. We then provide formal descriptions for several different classes of required behaviors that are commonly used in informal descriptions of spacecraft behavior. We also show how these formal descriptions can be combined using standard CSP algebraic operators to form system behavior specifications. Finally, we consider how a composite specification can be checked to ensure that the behavior it specifies matches the specifier’s intentions. We use the behavior of a notional scientific spacecraft as a running example.
4.1 Required Functions

The requirements for a spacecraft are usually specified in terms of the functions that the spacecraft is required to perform [20, 21, 104]. However, there is no uniform understanding of what constitutes a required function, and, in many cases, the “functional requirements” for a spacecraft include things that might more properly be considered specifications of behavior. In this section, we establish a formal definition of the term required function, and use this definition to draw a distinction between required functions and required behavior.

4.1.1 Defining Function

Most systems engineering standards define the term function using some variation of the phrase “an action the system must perform” [105]. The design research community uses a similar but more precise formulation, defining a function as “a relationship between inputs and outputs” [106]. Not surprisingly, this definition is related to the mathematical definition of a function as a map from elements of one set to elements of another set. Moreover, a requirement that a spacecraft perform a certain function can be viewed as a requirement that some set of inputs be mapped to some set of outputs. Viewing functions in this way allows a more formal definition of what we mean by the term function.

**Definition 5** (Required Function). A required function is a total, many-to-one binary relation from a set of inputs $X$ to a set of outputs $Y$:

$$f : X \rightarrow Y$$

Of course, required spacecraft functions are rarely expressed in the form of an input/output mapping. Instead, the mapping associated with a particular required function is left implicit in the statement of the requirement. However, by expressing required functions in terms of input/output mappings we can achieve a much more precise definition of what function is actually required.
Example 4.1.1. A spacecraft is required to change its attitude (or orientation) in response to commands from ground-based operators. More formally, we describe this function as the mapping

\[ f_{\text{Att}} : \text{AttitudeCommand} \to \text{Attitude} \]

where \text{AttitudeCommand} is a set of attitude commands, and \text{Attitude} is a set of spacecraft attitudes. This mapping establishes a requirement on the spacecraft by specifying the commands to which the spacecraft should respond, and identifying the corresponding attitude states that the spacecraft must produce in response to each command.

A simple attitude commanding requirement might be specified in terms of the function

\[ f_{\text{Att}} = \{(\text{detumble}, \text{sun} _\text{pointing}),\]
\[ (\text{science}\_\text{attitude}, \text{earth}\_\text{limb}\_\text{scan}),\]
\[ (\text{safe}\_\text{attitude}, \text{sun} _\text{pointing})\} \]

where

\[ \text{AttitudeCommand} = \{\text{detumble}, \text{science}\_\text{attitude}, \text{safe}\_\text{attitude}\} \]
\[ \text{Attitude} = \{\text{uncontrolled}, \text{sun} _\text{pointing}, \text{earth}\_\text{limb}\_\text{scan}\} \]

The values of \text{Attitude} are names representing qualitatively different ranges of attitude angles and rates.

More complex specifications might include attitude commands parameterized by the desired attitude, or values of \text{Attitude} expressed in terms of other reference objects (e.g. the position of the sun). A formal description of such complex specifications may be more easily represented in terms of a rule for relating inputs and outputs, rather than as an enumeration of all of the elements of the relation. Fortunately, the machine-readable CSP\textsubscript{M} incorporates a functional programming language with facilities for manipulating sets, sequences, and
tuples. This language can be used to formally represent both simple relational definitions, and more complex rule-based function definitions.

### 4.1.2 Function vs. Behavior

Required functions, as we have defined them, can precisely specify one aspect of what a spacecraft is required to do. However, that aspect does not include behavior. Functions describe what should happen, but leave open the question of the order in which those things should happen. We might assume that applying a function to two different input values “in sequence” will result in the corresponding output values appearing in the same sequence. But there is nothing in the semantics of functions that specifies the sequencing of outputs, or even defines the meaning of applying a function “in sequence.” The difficulties created by this lack of sequencing semantics are compounded when considering systems that involve multiple functions. For example, in some spacecraft applications it may be necessary to specify that the results of performing one function, such as the acquisition of a stable attitude, must happen before other spacecraft functions can occur. Requirements of this kind can be critical to defining the correct operation of a spacecraft. But they cannot be defined in terms of functions alone.

One approach to resolving the problem of specifying the sequencing of inputs and outputs is to express requirements in terms of mappings between sequences of inputs and sequences of outputs [107], rather than mappings from individual inputs to individual outputs. However, this approach can become cumbersome as the number of inputs and outputs increases. More importantly, it can obscure the individual input/output relationships captured by function specifications, and make it much more difficult to compose separate specifications.

An alternative approach to resolving the sequence specification problem is to draw a distinction between required input/output relationships that are independent of sequence (functions), and required sequences of inputs and outputs (behaviors). This approach resembles the traditional technique of arranging spacecraft functions in “functional flow block diagrams” [19–21] that specify the ordering of function executions. Taking this approach
allows mathematical representations which are better suited than functions to describing behavior, such as CSP process expressions, to be applied to the problem of specifying the sequencing relationships between different inputs and outputs. Furthermore, the input/output sequencing for a function can then be specified by associating a behavior with that function.

4.1.3 Lifting Function to Behavior

One way to associate a function specification with a behavior specification is to encapsulate the function inside a CSP process description, effectively lifting the function into the behavior domain. A simple lifted-function behavior might consist of a straightforward alternation of input and output values. This behavior can be generically defined as a parameterized process.

Definition 6 (LiftF). Given a required function

\[ f : X \rightarrow Y \]

the corresponding lifted function is a process

\[ \text{LiftF}(in, out, f) = in?x \rightarrow out!f(x) \rightarrow \text{LiftF}(in, out, f) \]

where \( in \) must be a channel of type \( X \), and \( out \) must be a channel of type \( Y \).

Example 4.1.3.1. The function specification \( f_{\text{Att}} \) (example 4.1.1.1) can be lifted to a process by defining input and output channels of the appropriate type,

\[ \text{channel cmd}_{\text{Att}} : \text{AttitudeCommand} \]
\[ \text{channel attitude} : \text{Attitude} \]

and instantiating the \( \text{LiftF} \) process with the function specification,

\[ \text{AttitudeCommanding} = \text{LiftF}(\text{cmd}_{\text{Att}}, \text{attitude}, f_{\text{Att}}) \]
The resulting *AttitudeCommanding* process can initially engage in any event in the set

\[
\{ \text{cmd}_{\text{Att}}.\text{detumble}, \text{cmd}_{\text{Att}}.\text{science\_attitude}, \text{cmd}_{\text{Att}}.\text{safe\_attitude} \}
\]

It then generates a corresponding *attitude* event, and awaits a new command.

Lifting a function specification to a process has two important effects. First, a lifted function explicitly defines the sequence in which function inputs and outputs must occur, which means that the lifted function is less ambiguous than its corresponding function specification. Second, a lifted function can be composed with other processes using standard CSP operators, allowing lifted functions to be combined descriptions of other aspects of spacecraft behavior in well-defined ways.

**Multiple Inputs and Outputs**

Some spacecraft functions require more than one input, or generate more than one output. Expressing such functions in terms of input/output mappings is a straightforward matter of defining the relevant input or output set as a Cartesian product of several input or output components. In the behavior domain, however, things become more complicated.

**Example 4.1.3.2.** A science measurement function that combines instrument measurements and the current attitude state into a data item can be specified as

\[
f_{\text{Sci}} : \text{Measurement} \times \text{Attitude} \to \text{ScienceData}
\]

Lifting this function definition into the behavior domain, we obtain

\[
\begin{align*}
\text{channel } & \text{in}_{\text{Sci}} : \text{Measurement} \times \text{Attitude} \\
\text{channel } & \text{downlink\_data} : \text{ScienceData} \\
\text{Science} = & \text{LiftF}(\text{in}_{\text{Sci}}, \text{downlink\_data}, f_{\text{Sci}})
\end{align*}
\]
The input channel of lifted function *Science* in example 4.1.3.2 has a type which is a Cartesian product of two input components. These two input components originate from different places: one is supplied by the scientific instrument, the other is read from the attitude state. An interface process is required to obtain the two input components from their separate sources, and combine them into an input tuple for the science function.

The need to define an interface process for the multi-input *Science* function exposes a new behavior specification issue related to functions. For any multi-input or multi-output lifted function, it is necessary to define the order in which the multiple inputs or outputs should be received or sent.

**Definition 7** (MI Behavior). An MI (Multi-Input) behavior is an interface process that specifies the acceptable ordering of inputs for a lifted multi-input function. An MI behavior aggregates multiple component input values into a tuple. An MI behavior \( P \) with input channels \( in_1, \ldots, in_n \) and output channel *tuple* has the following properties:

1. Tuple depends on most recent inputs
   \[ s \prec \langle \text{tuple},(a_1,\ldots,a_n) \rangle \in \text{traces}(P) \implies \forall i : \{1,\ldots,n\} \bullet \text{last}(s \downarrow in_i) = a_i. \]

2. One complete input round per tuple
   \[ s \in \text{traces}(P) \implies \forall i : \{1,\ldots,n\} \bullet ((s \downarrow \{in_i\}) - 1) \leq (s \downarrow \{\text{tuple}\}) \leq (s \downarrow \{in_i\}). \]

3. Liveness
   \[ (s,X) \in \text{failures}(P) \implies X \neq \bigcup_{i=1}^{n} \{in_i\} \cup \{ \text{tuple} \}. \]

**Definition 8** (MO Behavior). An MO (Multi-Output) behavior is an interface process that specifies the acceptable ordering of outputs for a lifted multi-output function. An MO behavior receives a tuple, and decomposes the tuple into component output values which are communicated to other processes in the appropriate order. An MO behavior \( P \) with input channel *tuple* and output channels \( out_1, \ldots, out_n \) has the following properties:

1. Outputs depend on most recent tuple
   \[ s \prec \langle out_i,x \rangle \in \text{traces}(P) \implies \exists (a_1,\ldots,a_n) \bullet \text{last}(s \downarrow \text{tuple}) = (a_1,\ldots,a_n) \land a_i = x. \]
2. One complete output round per tuple

\[ s \in \text{traces}(P) \implies \forall i : \{1, \ldots, n\} \bullet ((s \downarrow \{\text{tuple}\}) - 1) \leq (s \downarrow \{\text{out}_i\}) \leq (s \downarrow \{\text{tuple}\}). \]

3. Liveness

\[ (s, X) \in \text{failures}(P) \implies X \neq \bigcup_{i=1}^{n} \{\text{out}_i\} \cup \{\text{tuple}\}. \]

The MI and MO behavior definitions leave the door open for arbitrarily complex input and output sequencing. However, in practice MI and MO behaviors will most likely be either purely sequential or purely parallel.

**Example 4.1.3.3.** A purely sequential two-input behavior requires that a value is obtained from channel \(\text{in}_1\) before anything is read from channel \(\text{in}_2\).

\[ \text{SeqIn}_2(\text{in}_1, \text{in}_2, \text{tuple}) = \text{in}_1?x \rightarrow \text{in}_2?y \rightarrow \text{tuple}!(x, y) \rightarrow \text{SeqIn}_2(\text{in}_1, \text{in}_2, \text{tuple}) \]

Since the \(\text{SeqIn}_2\) process defined above is ready to receive new inputs as soon as it has sent an output tuple, the composite process that results from combining \(\text{SeqIn}_2\) with a \(\text{LiftF}\) process will be able to accept new inputs before it has output a value. In order to retain the strict input/output alternation semantics of the \(\text{LiftF}\) process it is convenient to define a MI/MO \(\text{LiftF}\) process which includes synchronization events that enforce the desired input/output alternation when composed with compatible MI and MO processes. These synchronization events can be hidden when not needed.

**Definition 9** (MIMO LiftF).

\[
\text{MIMOLiftF}(\text{in}, \text{out}, \text{reqin}, \text{ackout}, f) = \\
\text{reqin} \rightarrow \text{in}?x \rightarrow \text{out}!f(x) \rightarrow \text{ackout} \rightarrow \text{MIMOLiftF}(\text{in}, \text{out}, \text{reqin}, \text{ackout}, f)
\]

where \(\text{reqin}\) is requests a new input round from an MI behavior, and \(\text{ackout}\) signals the completion of an output round from an MO behavior.
Example 4.1.3.4. A purely parallel two-output behavior allows values to be output on channels $out_1$ and $out_2$ in an arbitrary order. It includes an acknowledgment channel that indicates to a $MIMOLiftF$ process when an output round is complete.

\[
ParOut_2(tuple, out_1, out_2, ackout) = \\
\text{let} \\
\quad Out_1 = tuple?(x, y) \rightarrow out_1!x \rightarrow ackout \rightarrow Out_1 \\
\quad Out_2 = tuple?(x, y) \rightarrow out_2!y \rightarrow ackout \rightarrow Out_2 \\
\text{within} \\
\quad Out_1 \parallel \{\{tuple, ackout\}\} \parallel Out_2
\]

Example 4.1.3.5. Continuing with the science measurement example, we assume that making a science measurement always precedes reading the attitude state. Given this assumption, the interface process can be expressed as a sequential MI behavior

\[
\text{channel } instrument : Measurement \\
\text{channel } sense\_attitude : Attitude \\
\text{channel } sci\_req, sci\_ack \\
IF_{Sci} = SeqIn_2(sci\_req, instrument, sense\_attitude, in_{Sci})
\]

Combining this process with a MIMO lifted function produces the full behavior associated with the spacecraft science measurement function (fig. 4.1):

\[
Science' = IF_{Sci} \\
\quad \parallel \{\{in_{Sci}, sci\_req\}\} \\
MIMOLiftF(in_{Sci}, downlink\_data, sci\_req, sci\_ack, f_{Sci}) \setminus \{sci\_ack\}
\]
4.2 Components of Spacecraft Behavior

Spacecraft requirements often include “functions” that do not fit the definition of required functions established in the previous section. In many cases, these required “functions” are actually behaviors. Based on a survey of spacecraft system requirements documents and design descriptions produced by various organizations [22–24,108–113], we have identified several classes of behavior commonly used in informal descriptions of spacecraft system-level behavior. In this section, we develop CSP interpretations of each of these classes of behavior. The resulting CSP processes allow spacecraft behaviors to be specified with greater precision and less ambiguity than the usual informal descriptions. Along with the lifted functions discussed in the previous section, the various processes defined in this section can be regarded as primitive components of spacecraft behavior, and used as building-blocks for the development of composite spacecraft behavior specifications.

4.2.1 Event Sequences

Perhaps the most straightforward type of spacecraft behavior specification is the linear sequence of events, an example of which is the excerpt from the *MGS Block Dictionary* [114] which appears in fig. 4.2. A generic event sequence specification is readily described in CSP using generalized sequential composition.

**Definition 10** (Event Sequence). An event sequence is a parameterized process

\[
\text{EventSeq}(S) = \quad e : S \cdot e \rightarrow \text{SKIP}
\]
The first event in the separation detect script is to select the high rate mode for the IMU . . . The second event is to arm the thrusters . . . Residual rates after the separation and yo-yo despin are then damped by commanding the spacecraft into the Despin/Deploy mode.

Fig. 4.2: Example of an event sequence.

where $S$ is a trace (list of events) $S = \langle e_1, \ldots, e_n \rangle$. The behavior of this process is to execute each event in the trace $S$, in the order in which it appears in the trace sequence.

**Example 4.2.1.1.** Following separation from a launch vehicle, a simple scientific spacecraft is required to execute a sequence of deployments in preparation for carrying out its mission. This requirement is described by the event sequence

\[
\text{channel } \text{deploy} : \{ \text{antenna}_1, \text{antenna}_2, \text{solar}_\text{array} \}
\]

\[
\text{Deployments} = \text{EventSeq}(\langle \text{deploy.solar}_\text{array}, \text{deploy.antenna}_1, \text{deploy.antenna}_2 \rangle)
\]

The behavior defined by this specification is equivalent to

\[
\text{Deployments} = \text{deploy.solar}_\text{array} \rightarrow \text{deploy.antenna}_1 \rightarrow \text{deploy.antenna}_2 \rightarrow \text{SKIP}
\]

4.2.2 State Transition Systems

Spacecraft requirements and design documentation often contain informal state diagrams similar to the example shown in fig. 4.3, and descriptions of transitions in spacecraft or subsystem states such as the example, excerpted from the *WIRE System Requirements* [24], which appears in fig. 4.4. Specifications of this kind are instances of state transition systems.
**Definition 11** (State Transition System). A *state transition system* is a parameterized process

\[
StateTransitions(s_0, \text{transition}, \text{TransitionDefs}) = \\
\text{let} \\
\quad \text{Transitions}(m) = \{(\text{event}, s') \mid (s, \text{event}, s') \in \text{TransitionDefs}\} \\
\quad \text{State}(s) = \\
\quad \quad \quad \Box(\text{event}, s') : \text{Transitions}(s) \bullet \text{event} \rightarrow \text{transition}.s' \rightarrow \text{State}(s') \\
\quad \text{within} \\
\quad \quad transition.s_0 \rightarrow \text{State}(s_0)
\]

where

- \(s_0\) is the state in which the transition system starts.
- \(\text{transition}\) is an auxiliary channel used to signal state transitions as events of the form \(\text{transition}.s\), where \(s\) is the state that results when the transition is complete.
- \(\text{TransitionDefs} \subseteq \text{State} \times \text{Event} \times \text{State}\) is a set of 3-tuples, each of which defines a transition in terms of a current state, triggering event, and corresponding new state.
• \( \text{Transitions}(s) \) is the set of event/new-state pairs for state \( s \).

• \( \text{State}(s) \) is a process that defines the transition behavior for state \( s \).

To ensure that the transition system is free of deadlock, it is required that
\[
\forall (\text{state}, \text{event}, \text{state}') \in \text{TransitionDefs} \bullet \text{Transitions}(\text{state}') \neq \emptyset.
\]

Example 4.2.2.1. The state diagram in fig. 4.3 defines a behavior in which transitions between different spacecraft modes are caused by the launch vehicle separation event, mode transition commands, and the occurrence of faults. This mode transition behavior can be expressed as a state transition system. For brevity, we assume that the transition trigger events can only occur in certain modes, and do not include responses to unexpected trigger occurrences in the transition behavior. The resulting spacecraft mode transition system, \( SC_{\text{Modes}} \), is

\[
\begin{align*}
\text{channel} & \ trans_{\text{Mode}} : \{\text{launch, safe, nominal}\} \\
\text{channel} & \ cmd_{\text{Mode}} : \{\text{safe, nominal}\} \\
SC_{\text{Modes}} & = \\
& \quad \text{let} \\
& \quad \quad \text{TransitionDefs} = \{(\text{launch, separation, safe}), \\
& \quad \quad \quad (\text{safe, cmd}_{\text{Mode}.\text{nominal}}, \text{nominal}), \\
& \quad \quad \quad (\text{nominal, cmd}_{\text{Mode}.\text{safe}}, \text{safe}), \\
& \quad \quad \quad (\text{nominal, fault, safe})\} \\
& \quad \text{within} \\
& \quad \quad \text{StateTransitions}(\text{launch, trans}_{\text{Mode}}, \text{TransitionDefs})
\end{align*}
\]

More complex kinds of transition system processes can be developed by modifying or expanding the definition given above. Alternatively, more complex behaviors can be added to a state transition system of the form defined above by composing the transition-system process with other processes. For example, outputs can be associated with the transition
system by composing it with one or more lifted functions that map state transitions to output values. Functions that take a single transition event as an input produce Moore-style outputs, while functions that take a pair of transitions or a transition/input pair result in Mealy-style outputs [39].

**Example 4.2.2.2.** Lifting the function

\[ f_{\text{ReportMode}} : \text{Mode} \rightarrow \text{DownlinkMsg} \]

and placing it in parallel with the \( SC_{\text{Modes}} \) process,

\[
\text{channel downlink} : \text{DownlinkMsg}
\]

\[
SC'_{\text{Modes}} = SC_{\text{Modes}} \parallel \langle \text{transMode} \rangle \parallel \text{LiftF(transMode, downlink, } f_{\text{ReportMode}}) \]

produces a specification \( SC'_{\text{Modes}} \) that calls for a message to be downlinked to the ground whenever a mode transition occurs.

### 4.2.3 Event-Triggered Behaviors

Some spacecraft designs require an event to occur before a behavior is activated. An example of such a requirement is the excerpt from the *MGS Spacecraft Requirements* [23] shown in fig. 4.5, in which a sequence of commands is triggered by the occurrence of an eclipse event. Event-driven triggers of this sort are straightforwardly expressed in terms of CSP’s prefixing operator. The more general case, in which several different events may trigger a given behavior, can be specified using generalized indexed choice.

**Definition 12** (Event-Triggered Behavior). An *event-triggered behavior* is a parameterized process

\[
\text{EventTrigger}(\text{Triggers}, P) = \square t : \text{Triggers} \cdot t \rightarrow P
\]

where \( \text{Triggers} \) is a set of triggering events, and \( P \) is a behavior to be triggered.
**Requirement 3.4.4.1.1** Upon eclipse entry the spacecraft shall initiate execution of a stored command sequence designed for eclipse ingress, and upon eclipse egress shall initiate execution of an independent stored command sequence designed for eclipse egress.

Fig. 4.5: Example of a triggered behavior requirement.

**Example 4.2.3.1.** The Deployments event sequence defined in example 4.2.1.1 should be initiated by spacecraft separation from the launch vehicle, signified by the separation event.

\[
\text{SeparationBehavior} = \text{EventTrigger}\{\text{separation}\}, \text{Deployments}\]

**Example 4.2.3.2.** Let IngressSeq and EgressSeq be the two command sequences mentioned in fig. 4.5. Then the requirement described in fig. 4.5 can be expressed as

\[
\text{EclipseSeqs} = \text{EventTrigger}\{\text{eclipse\_ingress}\}, \text{IngressSeq}; \\
\text{EventTrigger}\{\text{eclipse\_egress}\}, \text{EgressSeq}\}; \\
\text{EclipseSeqs}
\]

where the recursive sequential composition of the \text{EclipseSeqs} process indicates that the eclipse sequence behavior repeats for every eclipse.

**4.2.4 Defining Other Types of Behavior**

Although the different processes defined above represent the classes of behavior most commonly used in spacecraft requirements and design documentation, they are not the only types of behavior we might wish to specify. In fact, the ability to define arbitrary behaviors is one of the advantages of using a process algebraic approach to specification. Fortunately, since all of the behavior components described so far have been interpreted as CSP processes, they are fully compatible with any behaviors we might choose to define using arbitrary CSP processes, and can also be modified using standard CSP operators to produce new behaviors. This provides a great deal of flexibility in developing specifications.
In general, any CSP operator or construct can be used to specify a spacecraft behavior. However, it is necessary to impose two restrictions:

1. Each individual behavior description must be free of deadlock.
2. Each individual behavior description must be free of divergence.

The reason for the first restriction is that we wish to use deadlock to represent an error in the interactions between different specification components. Thus, deadlock should not be a deliberate part of a specification component. A specification component in a divergent state is no longer providing any information about observable events, and is therefore useless for defining observable spacecraft behavior.

In formal terms, the restrictions on the occurrence of deadlock and divergence amount to a requirement that a behavior description $P$ satisfy $DF^\triangledown \subseteq_{FD} P$, where

$$DF^\triangledown = (\bigcap e : \Sigma \bullet e \to DF^\triangledown) \cap SKIP$$

In practice, the restrictions with regard to deadlock and divergence mean that

1. No sequential behavior description can contain the $STOP$ process.

2. Any behavior description constructed from parallel processes must be proved free of deadlock and divergence (for example, through a refinement check using FDR).

**Example 4.2.4.1. Conjunction of events** Suppose that we wish to specify that a spacecraft with two propellant tanks will signal its ground station once both tanks are empty. There are a number of different ways to construct this specification. One way to specify the desired behavior is the following process:

\[
\begin{align*}
\text{SignalTanksEmpty} &= (\text{sense_empty}_{\text{tank}_a} \to \text{signal_empty} \to \text{SKIP}) \\
&\quad \square\{\text{signal_empty}\} \\
&\quad (\text{sense_empty}_{\text{tank}_b} \to \text{signal_empty} \to \text{SKIP})
\end{align*}
\]
This process will engage in the \textit{signal\_empty} event once both \textit{sense\_empty} events have occurred, following which it will terminate. The \textit{SignalTanksEmpty} process is simple enough that it can be determined to be deadlock-free by inspection, although it is good practice to confirm this determination using FDR.

\textbf{Example 4.2.4.2. Resettable LiftF} The behavior defined by the \textit{LiftF} process specifies that once a function is invoked it blocks until it can output a value. This may not always be a desirable behavior for a function. Suppose that we wish to define a class of lifted functions that can be reset before they have output their current value. This can be achieved using CSP’s interrupt operator:

\[
\text{ResettableLiftF}(in, out, f, \text{Reset}) =\\
\text{LiftF}(in, out, f) \triangle (\square r : \text{Reset} \cdot r \rightarrow \text{ResettableLiftF}(in, out, f, \text{Reset}))
\]

The \textit{ResettableLiftF} process behaves just like the corresponding \textit{LiftF}, except that whenever an event in the set \textit{Reset} occurs any pending output is forgotten, and the lifted function is instead ready to accept a new input.

\section*{4.3 Specifying Composite Spacecraft Behavior}

In many spacecraft design projects the specification of spacecraft behavior consists of nothing more than a few fragmentary descriptions of desired behavior. While the behavior components defined in previous sections can be used to write such fragmentary behavior specifications, the formal nature of the behavior components also make it possible to define the relationships and interactions between different behavior fragments.

The most straightforward relationship between two behavior components consists of parallel composition, and synchronization on common events. We have already seen several examples of this type of relationship in previous sections, such as the composition of a state transition system and a lifted function in example 4.2.2.2. In this section we consider two other, more indirect types of relationships between behaviors: interactions via shared resources, and constraints on the relative order of events belonging to different behaviors. We
also examine how the three different types of behavior relationships can be used together to create a composite specification defining the overall system-level behavior of the spacecraft.

4.3.1 Shared States

In many cases, the outputs generated by a particular spacecraft behavior component are influenced by some component of the current state of the spacecraft. That component of the spacecraft state may, in turn, be influenced by other behavior components. The dependencies between behaviors that are induced by a shared state component can thus have important implications for the overall behavior of a spacecraft. As a result, defining the state components which are shared between behaviors is a crucial part of developing a system-level spacecraft behavior specification.

Example 4.3.1.1. Examples 4.1.3.1 and 4.1.3.5 introduced an attitude commanding function, and a science data collection function. As part of its data collection behavior the science function senses the current attitude state. The value of the attitude state in turn depends on the attitude commanding function, which sets the spacecraft attitude in response to received commands. The attitude state is thus shared between these two behaviors.

Shared state components act as a form of memory. They record the cumulative activity of some behavior components, and make that record available to other behavior components. The record of cumulative activity may take various forms, depending on the relationship that exists between the behaviors that share a state. We represent shared state components as state-bearing processes. These shared state processes are ultimately composed in parallel with the behavior components that they influence, and by which they are influenced, thereby allowing the behavior components in question to indirectly interact.

Assignable States

Perhaps the simplest type of shared state is one which simply stores the most recent state value passed to it.

Definition 13 (Assignable State). An assignable state is a process parameterized by a state value \( val \), and three channels, \( set \), \( get \), and \( trans \). The process gives its environment
an external choice between assigning a new value of the parameter \textit{val} through the channel \textit{set}, or reading the current \textit{val} through the channel \textit{get}. Transitions in the state value are signaled through the \textit{trans} channel.

\[
\text{AssignableState}(\textit{set}, \textit{get}, \textit{trans}, \textit{val}) = \\
\textit{get}!\textit{val} \to \text{AssignableState}(\textit{set}, \textit{get}, \textit{trans}, \textit{val}) \\
\boxempty \\
\textit{set}?\textit{val}' \to \text{if } \textit{val}' \neq \textit{val} \\
\text{then } \textit{trans}!\textit{val}' \to \text{AssignableState}(\textit{set}, \textit{get}, \textit{trans}, \textit{val}') \\
\text{else } \text{AssignableState}(\textit{set}, \textit{get}, \textit{trans}, \textit{val})
\]

The definition of \textit{AssignableState} does not constrain the state values between which transitions are allowed to occur. That is, an assignable state does not impose any dynamics on the state values it holds. We assume that constraints on state transitions are a result of limitations in the capabilities of the behaviors that control the state, and are therefore captured as part of the behavior definition. The transition events generated by the assignable state process allow behavior components that depend on the state to be notified of a new value without having to continually read values from the state.

\textbf{Example 4.3.1.2.} The shared spacecraft attitude state of example 4.3.1.1 can be represented as an assignable state component

\[
\text{channel } \textit{attitude, sense\_attitude, attitude\_transition} : \textit{Attitude} \\
\text{AttitudeState } = \\
\text{AssignableState}(\textit{attitude, sense\_attitude, attitude\_transition, uncontrolled})
\]

\textit{AttitudeState} initially has the value \textit{uncontrolled}. The attitude commanding function may assign new values to the attitude state through the \textit{attitude} channel, resulting in a transition event being generated on the \textit{attitude\_transition} channel. The science function may read the current value of the attitude state through \textit{sense\_attitude}. ■
Quantitative Resources

Some types of behavior interact through shared use of a quantitative resource. While the value of an assignable state depends only on the last assignment to the state, the value of a quantitative resource depends on the order in which it has been operated upon. For example, the feasibility of executing a particular orbital maneuver may depend on the quantity of spacecraft $\Delta v$ that remains after previous maneuvers. Many quantitative resources can be modeled directly in terms of integer quantities, and those that are real-valued can often be scaled to fit an integer model. Integer-valued quantitative resources are better suited than real-valued quantities to analysis with presently available CSP tools.

Definition 14 (Integer Quantitative Resource). An integer-valued quantitative resource is a parameterized process

$$
\text{QuantResource}(\text{delta, get, trans, min, max, init}) =
$$

let

$$
\text{Range} = \{\text{min . max}\}$$

$$
\text{Quantity(val)} =
$$

$\text{val > max & qr\_exception.resource\_overflow \rightarrow STOP}$

$\square$

$\text{val < min & qr\_exception.resource\_underflow \rightarrow STOP}$

$\square$

$\text{val \in Range & get!val \rightarrow Quantity(val)}$

$\square$

$\text{val \in Range & delta?d \rightarrow let val' = val + d within}$

if $val' \neq val \land val' \in Range$

then trans!val' \rightarrow Quantity(val')

else Quantity(val')

within

$Quantity(\text{init})$
where

- *init* is the initial value of the quantity.
- *max* and *min* are the upper and lower bounds of the quantity.
- *delta* is a channel through which the value may be increased or decreased.
- *get* is a channel through which the current value may be read.
- *trans* is a channel through which changes in the value are signaled.
- *Range* is the set of all values in the interval defined by *max* and *min*.
- *Quantity(val)* is a process that defines the behavior of the resource for value *val*.
- *qr_exception* is a channel used to signal that the quantity is at a forbidden value.

The values of *max*, *min*, and *init* must be scaled such that the smallest possible increase or decrease in the quantity is 1.

**Example 4.3.1.3.** Consider a spacecraft with 30 m/s of total ∆v, consumable in increments of not less than 0.5 m/s. We can represent the quantity of ∆v as the quantitative resource

\[
\begin{align*}
  \text{min} &= 0 \\
  \text{max} &= 60 \\
  \text{channel} \ dv\_change & : \{-\text{max} .. \text{max}\} \\
  \text{channel} \ dv\_sense, dv\_trans & : \{\text{min} .. \text{max}\} \\
  \text{Available} \Delta v &= \text{QuantResource}(dv\_change, dv\_sense, dv\_trans, \text{min}, \text{max}, 60)
\end{align*}
\]

where the 30 m/s value for total ∆v has been scaled to 60 (i.e. \(\frac{30\text{m/s}}{0.5\text{m/s}}\)) in order to make the smallest change in ∆v equal to 1. Behaviors that involve an orbital maneuver may decrease the available ∆v by an amount *x* by synchronizing on the event *dv_change!(−x)*. Similarly, behaviors that involve reporting on the currently available ∆v can determine this value through the *dv_sense* channel. In the case of ∆v, synchronization on *dv_change* with positive values, which increases the amount of available ∆v, is unlikely to be used unless on-orbit refueling scenarios are being considered.
Buffers

Buffers provide a way to accumulate sequences of outputs produced by one or more behavior components for later input to other components. Use of a buffer as a shared state is appropriate when the behavior components that depend on the shared state are affected by both the individual values of events that alter the state, and the order in which those events occur.

Example 4.3.1.4. Consider a spacecraft command uplink function, and several behavior components that depend on the commands output by the uplink function. Assume that the uplink function is required to receive new inputs without waiting for any activities triggered by its outputs to be completed. Since the value of the command uplink outputs, and the order in which they are output, determines the response of the dependent behavior components, a buffer is an appropriate choice for defining the relationship between the uplink function and the dependent behavior components.

One of the simplest types of buffer is the bounded blocking buffer [3]. A bounded blocking buffer accepts only a limited number of values, after which it refuses further inputs until an output has occurred. A bounded blocking buffer also refuses to output when the buffer is empty.

Definition 15 (Bounded Blocking Buffer). A bounded blocking buffer is a parameterized process

\[
\text{BoundedBlockingBuffer}(\text{in}, \text{out}, N) =
\]

\[
\text{let}
\]

\[
\text{Buff}(s) = (\#s < N \& \text{ in}\,x \rightarrow \text{Buff}(s \triangledown \langle x \rangle))
\]

\[
\square
\]

\[
(\#s > 0 \& \text{ out}\,\text{head}\,s \rightarrow \text{Buff}(\text{tail}\,s))
\]

\[
\text{within}
\]

\[
\text{Buff}()\]

where \text{in} and \text{out} are input and output channels, respectively, and \(N\) is the bound or capacity of the buffer.
Example 4.3.1.5. The command buffer of example 4.3.1.4 might be modeled as a bounded blocking buffer capable of holding, for example, up to 20 commands:

\[
\text{channel } uplinked\_cmd, cmd : \text{Command} \\
\text{CommandBuffer} = \text{BoundedBlockingBuffer}(uplinked\_cmd, cmd, 20)
\]

Bounded blocking buffers are by no means the only type of buffer process. For some specifications, an alternative buffering behavior may provide a better description of the desired relationship between behavior components. Other possibilities for buffering behavior include:

- Providing for output of a subsequence of the buffered sequence, instead of a single buffered item.
- Allowing output, perhaps of the empty sequence or a special “empty” symbol, when the buffer is empty.
- Allowing new inputs to overwrite existing values when the buffer is full. There are several possible overwriting strategies.

Unfortunately, we do not have sufficient space here to develop process definitions for all of the possible buffering behaviors, but they are straightforward.

Other Types of Shared State

Selecting a particular type of shared state to define a relationship between different behavior components is a key step in defining an overall spacecraft behavior. While the simple types of state-bearing processes we have described in this section are probably sufficient for many specifications, they will not be appropriate in all cases. Fortunately, there are many other kinds of state-bearing processes which might be used to describe a particular desired relationship between behavior components. Any of these state-bearing processes which is both deadlock-free and divergence-free can be used to represent a shared state of some kind.
### 4.3.2 Constraints

Constraints are a general way to specify sequencing relationships between different behavior components. They represent an abstract specification of sequencing, rather than a concrete description of how the required sequence is actually produced.

**Example 4.3.2.1.** Example 4.1.3.5 introduced the Science behavior specification, which associates a scientific measurement with an attitude state value. We assume that it is desirable that attitude maneuvers do not interfere with the taking of a measurement. Although such a limitation on attitude maneuvers might be implemented in several different ways (e.g. centralized control of both maneuvers and measurements, or message-passing between independent science and attitude control subsystems) it can be simply and directly specified as a constraint on the occurrence of attitude state change events.

We represent constraints as processes which are placed in parallel with the behavior components that they constrain. The behavior components are required to synchronize on every event they share with a constraint process, which forces them to perform only those sequences of actions permitted by the constraints. We also permit constraints to include events which are not associated with any behavior component, since that allows greater flexibility in defining the relationships between constraints.

**Definition 16 (Constraint).** A constraint is a deadlock-free, divergence-free process which consists of events from the alphabets of one or more behavior components, shared states, or other constraint processes, and defines the acceptable sequencing of these events.

**Example 4.3.2.2.** The generic Between and Outside constraint processes allow events in the set $E$ to occur only in the interval between the occurrence of some enabling event, $en \in \text{Enable}$, and the next occurrence of a disabling event $dis \in \text{Disable}$. The difference between the two constraints is that Between assumes that events in $E$ are initially disabled, while Outside assumes that they are initially enabled (see fig. 4.6). It is further assumed that the enabling and disabling event sets are disjoint, i.e. $\text{Enable} \cap \text{Disable} = \emptyset$. 

The **Between** and **Outside** constraint processes can be defined in terms of each other [3]:

\[
\text{Between}(\text{Enable}, \text{Disable}, E) = \\
(\square \text{en} : \text{Enable} \bullet \text{en} \rightarrow \text{Outside}(\text{Disable}, \text{Enable}, E)) \\
\square
\]

\[
(\square \text{dis} : \text{Disable} \bullet \text{dis} \rightarrow \text{Between}(\text{Enable}, \text{Disable}, E))
\]

\[
\text{Outside}(\text{Disable}, \text{Enable}, E) = \\
(\square e : E \bullet e \rightarrow \text{Outside}(\text{Disable}, \text{Enable}, E)) \\
\square
\]

\[
(\square \text{en} : \text{Enable} \bullet \text{en} \rightarrow \text{Outside}(\text{Disable}, \text{Enable}, E)) \\
\square
\]

\[
(\square \text{dis} : \text{Disable} \bullet \text{dis} \rightarrow \text{Between}(\text{Enable}, \text{Disable}, E))
\]

---

**Example 4.3.2.3.** The attitude maneuvering constraint informally described in example 4.3.2.1 can be formally captured using an **Outside** constraint process:

\[
\text{SciAttConstr} = \text{Outside}([\text{instrument}], [\text{downlink\_data}], [\text{attitude}])
\]

This specification disables attitude state change events between any instrument event and the next downlink_data event.
Mode Constraints

A system that has multiple possible behaviors may not always be ready to engage
in all of those behaviors. We say that such systems exhibit different modes of behavior.
Figure 4.7 provides an example of an informal definition of the behavior of a spacecraft in
different modes.

A state transition system, such as the one described in example 4.2.2.1, can be used to
define the mode transition behavior, in terms of the events that trigger mode transitions.
However, a mode transition behavior by itself does not define which other behaviors may
be exhibited in different modes. We specify the relationship between the mode transition
behavior and the behaviors that are enabled and disabled in different modes using a mode
constraint process, which provides a formal equivalent of the kind of informal mode de-
scriptions exemplified by the excerpt from the EO-1 Spacecraft-to-Ground ICD [115] which
appears in fig. 4.7. A complete specification of the modes of behavior for a spacecraft system
consists of both a mode transition behavior, and a collection of mode constraints.

Definition 17 (Mode Constraint). A mode constraint is a constraint process

\[
\text{ModeConstraint}(\text{InitEv}, \text{transition}, \text{Enabled}, \text{Disabled}) =
\]

\[
\text{let}
\]

\[
\text{En} = \{\text{transition}.m \ | \ m \in \text{Enabled}\}
\]

\[
\text{Dis} = \{\text{transition}.m \ | \ m \in \text{Disabled}\}
\]

\[
\text{within}
\]

\[
\text{Between(En, Dis, InitEv)}
\]

where

- \text{InitEv} is the set of initial events for some behavior component.
- \text{transition} is a channel used to signal mode transitions.
- \text{Enabled} is the set of modes in which behavior initiated by events in \text{InitEv} is enabled.
3.6 EO-1 MODES OF OPERATION
This section describes the spacecraft communications configuration in various mission modes.

3.6.1 LAUNCH COMMUNICATIONS CONFIGURATION
...EO-1 will begin transmitting 2-kbps S-band telemetry data to TDRSS 30 seconds after fairing separation.

3.6.2 NORMAL OPERATIONS MODE COMMUNICATIONS
For normal operations, the EO-1 spacecraft will not transmit telemetry routinely. Telemetry downlinks will be planned, coordinated with ground stations, and initiated by real-time or stored command.

3.6.3 BACKUP SCIENCE MODE
In case of communication problems with the X-band science downlink, a backup S-band science communications mode is implemented.

3.6.4 SAFE MODE COMMUNICATIONS
TBD

Fig. 4.7: EO-1 spacecraft behavior in different modes.

- **Disabled** is the set of modes in which behavior initiated by events in InitEv is not enabled.

- **Enable \( \cap \) Disable = \( \emptyset \).**

**Example 4.3.2.4.** A simple scientific spacecraft has three modes of behavior: an initial launch mode (no functions), a safe mode (attitude control, but no science), and a nominal mode (attitude control and science). The mode transition behavior for the spacecraft is defined by the \( SC_{\text{Modes}}^t \) transition system described in example 4.2.2.1. The corresponding mode constraints are:

\[
\text{Attitude}_{\text{Modes}} = \text{ModeConstraint}(\{\text{cmd}_{\text{Att}}\}, \text{trans}_{\text{Mode}}, \{\text{safe, nominal}\}, \{\text{launch}\})
\]

\[
\text{Science}_{\text{Modes}}^t = \text{ModeConstraint}(\{\text{instrument}\}, \text{trans}_{\text{Mode}}, \{\text{nominal}\}, \{\text{launch, safe}\}),
\]

where \( \text{trans}_{\text{Mode}} \) is the channel used to signal mode transitions.

Collectively, the mode transition behavior \( SC_{\text{Modes}}^t \), along with the mode constraints \( \text{Science}_{\text{Modes}}^t \) and \( \text{Attitude}_{\text{Modes}} \), specify that:
• The spacecraft is initially in the launch mode.

• While in the launch mode the spacecraft does not perform any function.

• Upon launch-vehicle separation the spacecraft transitions to safe mode.

• While in safe mode the attitude can be commanded, but no science can be performed. The spacecraft transitions to nominal mode upon command.

• While in nominal mode the spacecraft can respond to attitude commands and take instrument measurements. The spacecraft transitions to safe mode upon command.

• The occurrence of fault events during nominal mode causes the spacecraft to revert to safe mode.

**Functional Flow Block Diagrams as Constraints**

Functional Flow Block Diagrams (FFBDs) are a traditional tool of spacecraft systems engineering [20, 21, 104]. Classical FFBDs focus purely on function sequencing [116]. As a result, they can, within the context of the framework developed in this chapter, be viewed as constraints on the sequencing of lifted functions.

Although it is possible to directly construct a constraint process to represent a given FFBD, such a construction is likely to be difficult for any non-trivial diagram. Fortunately, the basic graphical elements of FFBDs (fig. 4.8) are readily described in terms of CSP processes, which makes it possible to construct a constraint-process representation of an FFBD in a compositional manner. However, the exact meaning attached to the different FFBD elements varies somewhat from author to author. Here we present one possible interpretation of the semantics of FFBDs, defined as a mapping from each FFBD element to a corresponding CSP process.

**Definition 18 (FFBD Process).** An FFBD process is an FFBDblock process, or a composition of FFBDblock processes formed using the FFBDseq, FFBDand, FFBDor, FFBDchoice, and FFBDiteration compositions.
The fundamental primitive from which FFBDs are constructed is the function block, which is simply a box labeled with a function name. We assume that function blocks denote a single execution of the function represented by their label.

**Definition 19 (FFBD Block).** An *FFBD block* is a terminating process parameterized by the input and output channels of the function represented by the block.

\[
FFBDBlock(in, out) = \Box i : \{in\} \cdot i \rightarrow (\Box o : \{out\} \cdot o \rightarrow SKIP)
\]

An *FFBD block* constraint process simply permits a single input/output cycle of a lifted function. It does not constrain the *values* that the function can input or output. In the case of a MIMO lifted function (see sec. 4.1.3), we use the *reqin* and *ackout* channels to define the FFBD block associated with the function, since those channels provide a single point of control over the *MIMOLiftF* inputs and outputs.
Example 4.3.2.5. The function block $Function_1$, with input $in_1$ and output $out_1$, corresponds to the process

$$Function_1 = FFBD\text{block}(in_1, out_1)$$

The FFBD sequencing notation translates directly to sequential composition in CSP.

Definition 20 (FFBD Sequence). An FFBD sequence is a process parameterized by two FFBD processes to be executed in sequence.

$$FFBDseq(FFBD_1, FFBD_2) = FFBD_1; FFBD_2$$

The FFBD concurrent composition notation indicates that each of the branches emanating from the “AND” bubble are executed concurrently, with the entire concurrent composition terminating when all of the branches have terminated. The AND relationship thus maps well to the CSP interleaving operator.

Definition 21 (FFBD Concurrent Composition). An FFBD concurrent composition is a process parameterized by a set of FFBD processes to be executed concurrently.

$$FFBD\text{and}(FFBD\text{set}) = || FFBD : FFBD\text{set} \cdot FFBD$$

The FFBD selection notation indicates that each of the branches emanating from the “OR” bubble is a valid alternative behavior. The method of selecting an alternative is often left unspecified. We assume that the choice is resolved by the occurrence of an initial event from one of the OR branches, which allows the OR relationship to be represented as a generalized external choice.

Definition 22 (FFBD Selection). An FFBD selection is a process parameterized by a set of FFBD processes, one of which may be selected for execution.

$$FFBD\text{or}(FFBD\text{set}) = \Box FFBD : FFBD\text{set} \cdot FFBD$$

The FFBD choice notation specifies which of two branches may be taken, based on the outcome of some test. We assume that the test is performed on the value of some shared
state, and that some subset of the shared-state values corresponds to a decision to take one branch, while the complementary set of values results in the other branch being taken.

**Definition 23 (FFBD Choice).** An **FFBD choice** is a process parameterized by the output channel of a shared state process, a set of values corresponding to the Go branch, and two FFBD processes.

\[
\text{FFBDchoice}(\text{test\_state}, \text{GoSet}, \text{GoFFBD}, \text{NoGoFFBD}) = \\
\text{test\_state?val} \rightarrow \text{if } \text{val} \in \text{GoSet} \text{ then GoFFBD else NoGoFFBD}
\]

FFBD iteration allows a portion of an FFBD to be repeated until some condition is met. It is readily defined in terms of the **FFBDseq** and **FFBDchoice** processes.

**Definition 24 (FFBD Iteration).** An **FFBD iteration** is a process parameterized by the output channel of a shared state process, a set of values corresponding to the termination of the iteration, and an FFBD process to be repeatedly executed.

\[
\text{FFBDiteration}(\text{test\_state}, \text{GoSet}, \text{FFBD}) = \\
\text{let} \\
\text{Loop} = \text{FFBDseq}(\text{FFBD}, \text{FFBDchoice}(\text{test\_state}, \text{GoSet}, \text{SKIP}, \text{Loop})) \\
\text{within} \\
\text{Loop}
\]

**Example 4.3.2.6.** Figure 4.9 depicts a simple FFBD. Using the FFBD processes defined above, this simple FFBD can be translated directly to CSP.

For the purposes of this example, we assume that the FFBD functions operate on the dummy channels \(in_1 \ldots in_5\) and \(out_1 \ldots out_5\). We further assume that the state tested by the FFBD choice construct is also represented by a dummy channel, \(test\_state\). The resulting CSP process description is:
channel $in, out : \{1 \ldots 5\}$
channel $test\_state : \{0, 1\}$

$Function(n) = FFBD\text{block}(in.n, out.n)$

$SimpleFFBD = FFBD\text{iteration}(test\_state, \{1\},$

\[
FFBD\text{seq}(Function(1),$

\[
FFBD\text{and}(\{Function(2),$

\[
Function(3),$

\[
FFBD\text{or}(\{Function(4),$

\[
Function(5)\})))$)

Temporal Constraints

The framework developed in this chapter is primarily concerned with event ordering. However, constraint processes can also be used to specify the timing of events. These specifications require the introduction of a timeline, defined in terms of *temporal events*.

**Definition 25** (Temporal Event). A *temporal event* is an auxiliary specification event representing an instant of time that has some significance for the spacecraft behavior.
Definition 26 (Timeline). A timeline is a parameterized process

\[ \text{Timeline}(T) = \; ; \; t : T \cdot t \rightarrow \text{SKIP} \]

where \( T \) is a list of temporal events, \( T = \langle t_1, \ldots, t_n \rangle \).

Example 4.3.2.7. The timeline for the example timing requirement shown in fig. 4.10, excerpted from the WIRE System Requirements [24], might be defined as

\[ \text{SepTimeline} = \text{Timeline}(\langle \text{sep\_plus\_10sec}, \text{sep\_plus\_12sec}, \text{sep\_plus\_40min} \rangle) \]

where \( \text{sep\_plus\_10sec}, \text{sep\_plus\_12sec}, \) and \( \text{sep\_plus\_40min} \) are temporal events.

A timeline process establishes a sequence of temporal events. The timing of system events is specified in terms of constraint relationships between system events and temporal events. A complete timing specification is produced by synchronizing the timeline process and all of the temporal constraints on the temporal events. Defining a timeline consisting of a single, infinitely repeating temporal event (e.g. tock) results in the standard CSP model of discrete time described in Roscoe [3].

Temporal constraints define intervals of a timeline over which certain system events are permitted to occur. Since CSP events are always assumed to be interleaved, i.e. no two events can occur simultaneously, specifying that a particular system event must occur at a

<table>
<thead>
<tr>
<th>Time After Separation</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 sec</td>
<td>ACE box on</td>
</tr>
<tr>
<td>12 sec</td>
<td>Solar Array Deploy</td>
</tr>
<tr>
<td>40 min</td>
<td>Secondary hydrogen vent open</td>
</tr>
<tr>
<td>40 min</td>
<td>Primary hydrogen vent open</td>
</tr>
</tbody>
</table>

Fig. 4.10: Example of an event timing requirement.
given time requires the creation of two temporal events that bound the time interval within which the system event must occur.

**Example 4.3.2.8.** A specification that a set of events *may* occur in the interval between two times is easily constructed using the *Between* process:

\[
MayBetweenTimes(Events, time_1, time_2) = Between(\{time_1\}, \{time_2\}, Events)
\]

**Example 4.3.2.9.** A specification that a set of events *may* occur before some time can be constructed using the *Outside* constraint. A dual constraint, which permits events to occur only after some time, can be constructed using the *Between* process.

\[
MayBeforeTime(Events, time) = Outside(\{time\}, \emptyset, Events)
\]
\[
MayAfterTime(Events, time) = Between(\{time\}, \emptyset, Events)
\]

In both cases, the empty set is used to signify that the temporal constraints do not define a time at which the constrained events may later be re-enabled (in the case of *MayBeforeTime*) or disabled (in the case of *MayAfterTime*).

Any type of constraint process may be used as a temporal constraint, although the construction of temporal constraints can require some care to ensure that the specified behavior matches the desired behavior. A good discussion of these issues in the context of the standard CSP discrete-time model can be found in Chapter 14 of [3].

**Example 4.3.2.10.** One way to specify that a system event *must* occur before some time is to require that the temporal event corresponding to the time in question cannot occur before the system event:

\[
MustBeforeTime(event, time) = event \rightarrow time \rightarrow SKIP
\]
This specification will not allow the timeline to proceed until *event* has occurred. However, it also allows only a single occurrence of *event*, which may or may not be the intent of the specifier.

Example 4.3.2.11. Returning to the timeline defined in example 4.3.2.7, the temporal constraint on the “ACE box on” event might be specified as

\[
ACEOnTime = MayAfterTime(\{ ace\_box\_on \}, sep\_plus\_10\text{sec}) \quad \parallel \\{ ace\_box\_on \}\]

\[
MustBeforeTime(ace\_box\_on, sep\_plus\_12\text{sec})
\]

where for simplicity we have assumed that the requirement is simply that the ACE box event must occur before the next event in the timeline.

The *ACEOnTime* specification states that the *ace\_box\_on* event can only occur after time *sep\_plus\_10\text{sec}*, and *must* occur before time *sep\_plus\_12\text{sec}*. The parallel composition of the two temporal constraints forces both constraints to synchronize on the *ace\_box\_on* event, ensuring that both constraints are applied simultaneously.

4.3.3 Composite Behavior Specifications

A system-level spacecraft behavior specification is an integrated description of the observable behavior of a spacecraft. One way to construct a spacecraft behavior specification is as a composition of various fragmentary behavior descriptions, each of which provide a different view of the overall spacecraft behavior. The behavior descriptions may be individual behavior components, such as those described in sections 4.1 and 4.2, or may themselves be composite behavior specifications, such as a composition of lifted functions with an associated FFBD constraint. Direct interactions between different behavior descriptions, such as dataflow between functions, can be expressed using CSP’s parallel composition operators. Constraints and shared states provide a way to define indirect interactions between different fragments of behavior. A composite specification is constructed by combining behavior descriptions, constraints, and shared states in parallel.
The compositional approach to specification construction described here is similar to the constraint-oriented style of specification [117]. Behavior components can be viewed as “local constraints,” while shared states and constraint processes can be viewed as “end-to-end constraints.” However, unlike the constraint-oriented style, our compositional style also permits the introduction of specification events which are not part of the concrete observable system behavior, but are useful for developing specifications (e.g. temporal events).

In the interests of specification clarity, we first aggregate all of the specification elements of a particular type, and then combine these aggregate processes into a complete specification. Behavior components may directly interact via synchronization on certain shared events (e.g. state transition signals from shared state processes), but otherwise operate completely independently. As a result, the behavior components are aggregated through interface parallel composition, with each interface set defining the direct interactions.

**Definition 27** (Aggregate Spacecraft Behaviors).

\[
SpacecraftBehaviors = (BehaviorComp_1 \| IF_1 \| BehaviorComp_2) \| IF_2) \| \ldots \| IF_{(n-1)} \| BehaviorComp_n
\]

where each \(BehaviorComp_i\) is a behavior component, and each set \(IF_i\) specifies any direct interactions between the first \(i\) behavior components and component \(i+1\).

In most cases, separate behavior components will be independent of one another, with the result that \(IF_i = \emptyset\) and the interface parallel composition reduces to interleaving.

Shared state processes are all assumed to be completely independent of each other, and are therefore aggregated using the interleaved parallel operator.

**Definition 28** (Aggregate Shared States).

\[
SharedStates = || i : \{1 \ldots p\} \bullet State_i
\]

where each \(State_i\) is a shared state process.
In contrast to shared states, constraint processes are not independent, since different constraint processes may simultaneously impose restrictions on the same event. We therefore use the generalized alphabetized parallel operator to define a multi-process synchronization that conjoins multiple constraints by requiring every constraint that has a given event in its alphabet to synchronize on that event.

**Definition 29** (Constraint Network).

\[\text{ConstraintNet} = \| i : \{1 \ldots q\} \bullet \alpha(\text{Constraint}_i) \circ \text{Constraint}_i,\]

where each \(\text{Constraint}_i\) is a constraint process with alphabet \(\alpha(\text{Constraint}_i)\).

A complete composite specification combines the behaviors, shared states, and constraints through an interface parallel composition which allows the spacecraft behaviors to operate on the shared state, and the constraints to limit the spacecraft behaviors.

**Definition 30** (System Behavior Specification). A spacecraft system behavior specification is a composite process

\[\text{SysSpec} = (\text{SpacecraftBehaviors}[\| IF_{States} \| \text{SharedStates}]) [\| IF_{Constraints} \| \text{ConstraintNet}]\]

where

- \(IF_{States} = \alpha(\text{SpacecraftBehaviors}) \cap \alpha(\text{SharedStates})\) is the interface between the spacecraft behaviors and the shared state processes.

- \(IF_{Constraints} = \bigcup_{i=1}^{q} \alpha(\text{Constraint}_i)\) is the interface between the constraints, and the behaviors and states.

The system behavior specification composition is defined such that the behavior components do not have to participate in every shared state event, which permits the aggregate spacecraft behavior process to ignore shared state events, such as transition signals, if they
are not relevant to the behavior. In contrast, the behaviors are required to synchronize on every event in the alphabet of the constraint network.

**Example 4.3.3.1.** We can combine the behavior components defined in earlier examples to produce a composite behavior specification for a scientific spacecraft. In this simple example, the behavior consists of just two lifted functions, a single state transition system, and a single event sequence.

\[
\text{SpacecraftBehaviors} = \\
((\text{AttitudeCommanding} \parallel \text{Science'}) \parallel \text{SC'Modes}) \parallel \{\text{separation}\} \parallel \text{SeparationBehavior}
\]

There is only a single shared state, representing the possible interaction between the attitude function and the science function.

\[
\text{SharedStates} = \text{AttitudeState}
\]

The three constraints are the mode constraints for the attitude and science functions, and the constraint on interactions between the two functions.

\[
\text{ConstraintNet} = \parallel k : \{1..3\} \bullet \alpha_k \circ \text{Constr}_k
\]

where

- \( \text{Constr}_1 = \text{SciAtt}_{\text{Constr}} \) with \( \alpha_1 = \{\text{instrument, downlink_data, attitude}\} \).
- \( \text{Constr}_2 = \text{Attitude}_{\text{Modes}} \) with \( \alpha_2 = \{\text{cmd}_{\text{Att}}, \text{trans}_{\text{Mode}}\} \).
- \( \text{Constr}_3 = \text{Science'}_{\text{Modes}} \) with \( \alpha_3 = \{\text{instrument, trans}_{\text{Mode}}\} \).
The composite specification is:

\[ SC_{Spec} = (SpacecraftBehaviors \\
\quad \lvert\{\text{attitude, sense\_attitude}\}\rvert \\
\quad \text{SharedStates} \\
\quad \lvert\{\text{instrument, downlink\_data, attitude, cmd\_Att, trans\_Mode}\}\rvert \\
\quad \text{ConstraintNet} \]

**Example 4.3.3.2.** Assume that it is decided that the occurrence of fault events should result in a report of the fault to ground station, and a transition to a sun-pointing attitude. One way to specify this fault response is to define a new behavior

\[ FaultResponse = \\
\quad \text{EventTrigger}\{\text{fault}\}, \\
\quad \text{EventSeq}(\langle\text{downlink.fault\_occurred, attitude.sun\_pointing}\rangle); \text{FaultResponse} \]

This behavior is easily added to the composite specification by creating a new aggregation of spacecraft behaviors that extends the existing SpacecraftBehavior process with the new FaultResponse process. In making this extension it is necessary to synchronize FaultResponse on the fault event, since the spacecraft mode transition definition also depends on that event. The resulting aggregate behavior is

\[ SpacecraftBehaviors' = SpacecraftBehaviors \parallel \{\text{fault}\} \parallel \text{FaultResponse} \]

The new SpacecraftBehaviors' process can be substituted for SpacecraftBehaviors in the definition of \( SC_{Spec} \) to create a new composite specification.
4.3.4 Checking Composite Behavior

Composite spacecraft behavior specifications are CSP processes, and are readily translated into CSP$_M$, the machine-readable dialect of CSP. To ease the transition to CSP$_M$, we have created a library of CSP$_M$ process descriptions based on the definitions in this chapter. These process descriptions can be used to build machine-readable composite specifications. The CSP$_M$ spacecraft behavior library is listed in appendix A. Machine-readable spacecraft behavior specifications can be analyzed using several different commercially available tools. In this section we consider some of the basic checks that can be performed on a specification. We defer discussion of other, more involved verification activities to the next chapter.

Exploring Possible Behaviors

Tools such as the ProBE process behavior explorer [49] can be used to step through the possible executions of a behavior specification, examining the events that are possible at each step. Stepping through execution traces in this way allows specifiers to quickly gain an understanding of the emergent behavior defined by the specification. The capability to explore and understand individual spacecraft behavior components, as well as the interactions between different behavior components, is not available using traditional, informal specification techniques. CSP-based specifications thus provide spacecraft designers with an opportunity to detect undesirable behaviors at the time of specification, instead of waiting until problems manifest themselves during system integration.

Example 4.3.4.1. A translation into CSP$_M$ of the composite specification defined in example 4.3.3.1, performed using the spacecraft behavior library, appears in appendix B. Stepping through the CSP$_M$ version of the composite specification using ProBE, we can observe the following trace:

```
trans_Mode.launch
downlink.modestatus.launch
separation
deploy.solar_array
```
This trace shows that the specification, as written, permits a downlink to occur prior to separation from the launch vehicle. This is unlikely to be a desirable behavior from the perspective of a launch vehicle provider. To remove this behavior from the specification, we might add an additional constraint

$$\text{NoDownlinkBeforeSep} = \text{Between}(\{\text{separation}\}, \emptyset, \{\text{downlink}\})$$

This constraint disables events on the downlink channel until after the separation event has occurred.

**Checking Specification Consistency**

Beyond simple exploration of possible execution traces, it is also possible to perform automatic checks to ensure that a composite specification is consistent. Intuitively, a consistent composite specification is one in which the processes that make up the specification do not contradict each other. Consistency checks thus provide a basic tool for ensuring that every possible execution of a composite specification “makes sense.”

We define a consistent composite specification as a specification that is always able to make some kind of progress. In other words, a consistent specification is free from global contradictions: it can never reach a state in which none of the processes that make up the specification can agree on how to proceed. This definition of consistency corresponds to freedom from deadlock.

**Definition 31** (Consistent Composite Specification). A **consistent composite specification** is a composite specification that is free of deadlock.
Automated consistency checking can be accomplished as a test for deadlock using a tool such as FDR.

**Example 4.3.4.2.** Although FDR provides a built-in deadlock test, this test does not account for successful termination. Since successful termination is permitted within our framework, deadlock testing of composite specifications should generally be performed as a refinement check using the $DF^\lhd$ process. To check the CSP$_M$ composite specification that appears in appendix B for consistency, we add a definition of $DF_{tick}$, and an associated refinement assertion, to the CSP script:

$$DF_{tick} = (\|\| e:Events \& e \rightarrow DF_{tick}) \|\| SKIP$$

assert $DF_{tick}$ [FD= SCspec]

FDR confirms that SCspec failures/divergence refines $DF_{tick}$, which indicates that SCspec is a consistent specification.

**Example 4.3.4.3.** Assume that we add a constraint that prevents downlinking while the spacecraft is in the *uncontrolled* attitude. This constraint might, for example, reflect some physical limitation of the proposed antenna system which prevents it from guaranteeing communications in all possible attitudes. The corresponding constraint process is

$$NoDLWhileTumbling = Between\{attitude\_transition.sun\_pointing, attitude\_transition.earth\_limb\_scan\}, \{attitude\_transition.uncontrolled\}, \{downlink\}$$

With this constraint in place, performing a consistency check using FDR indicates that the specification is inconsistent. An investigation of the results generated by FDR indicates that the inconsistency results from the fact that the new constraint requires the achievement of a controlled attitude before a downlink can occur, while the rest of the specification requires a downlink of mode status information to occur before the spacecraft can enter a mode in which it can be commanded to detumble. There are several possible solutions to this problem, including removing the constraint, altering the requirement to downlink mode information, and adding a behavior to autonomously detumble the spacecraft after
separation from the launch vehicle. Which of these solutions is the correct one will depend on the mission of the spacecraft.

Further examples of using FDR to check and correct system-level spacecraft behavior specifications built within the framework described in this chapter can be found in the conference paper “A Model-Based Design Tool for Systems-Level Spacecraft Design” [80]. The complete CSP$_M$ for the examples discussed in that paper can be found in appendix C.

4.4 Summary

Strictly functional specifications define what a spacecraft is intended to do, but do not provide clear information on the order in which those things should be done. However, in many cases the sequencing of spacecraft inputs, outputs, and state changes is crucial to correct mission operations. This essential sequencing information is defined in a behavior specification. By using a formal approach to expressing behavior specifications we can precisely and unambiguously describe the desired system-level spacecraft behavior.

The conceptual framework developed in this chapter allows system-level behavior specifications to be developed in a modular and systematic manner. The different elements of the framework codify common spacecraft behavior terms, concepts, and specification patterns. Because this unified framework is defined in terms of CSP it can be readily extended with new CSP expressions that describe behavior not easily captured using the various behavioral elements defined in this chapter. Specifications developed within the framework provide an integrated description of spacecraft systems-level behavior not previously available to spacecraft designers. Furthermore, these integrated specifications can be analyzed using existing CSP tools, helping spacecraft designers to understand the composite behavior they have specified. A prototype tool for translating graphical mode transition diagrams and FFBDs into the constructs defined within this framework has been developed by Dr. Brandon Eames and Jared Crace [80].
Chapter 5
Modeling Spacecraft Subsystem Interactions

“It is necessary to study not only parts and processes in isolation, but also to solve the decisive problems found in organization and order unifying them...”
– Ludwig von Bertalanffy

The behavior of a spacecraft system emerges from the interactions between the subsystems that make up the spacecraft. Spacecraft designers must define the spacecraft subsystems, and the ways in which those subsystems interact, such that the required system-level behavior is generated. However, the tools presently used to accomplish this task are fairly limited. System block diagrams, such as the example in fig. 5.1, are often used to illustrate how a spacecraft system is composed from its subsystems, but do not define the behavior of the subsystem blocks, and cannot be used to infer the behavior of the system. Subsystem specifications may contain partial descriptions of subsystem behavior, such as those found in the ACE Spacecraft Design Specification [22], the MGS Spacecraft Requirements [23], and the WIRE System Requirements [24], but those descriptions are typically informal, and not suitable for deriving a rigorous understanding of the behavior that results when different subsystems interact. Existing approaches to mathematical modeling of subsystem interactions focus on building dynamic models of resource consumption [27,28], and do not address subsystem behavior in response to commands, events, and qualitative changes in subsystem state. In this chapter, we show how the behavior of different spacecraft subsystems can be formally modeled using CSP, and how these subsystem models can be composed into a spacecraft system model suitable for exploring the event-driven behavior of the system. We begin with an example which illustrates the motivation for developing this new approach to modeling spacecraft subsystems and systems.
5.1 A Motivating Example

Consider a spacecraft attitude control subsystem which is assumed to have the following simple behavior:

- When the attitude control subsystem is not powered, the spacecraft attitude is uncontrolled.
- When power is supplied to the attitude control subsystem, the attitude control system is able to make spacecraft attitude changes in response to received commands.
- When power is supplied to the attitude control system, every attitude command results in the commanded attitude being achieved.
- Switching off the supply of power to the attitude control subsystem causes the attitude to again become uncontrolled.
The behavior described above seems fairly straightforward, bordering on “common sense.” The description itself provides a much more explicit definition of the assumed attitude control subsystem behavior than would usually be found in a spacecraft requirements or specification document. It may not immediately be clear why we should bother with describing such a simple behavior in this kind of detail, let alone go to the trouble of constructing a process model to represent the behavior. However, a little reflection on the description given above reveals that it contains some ambiguities which have the potential to produce problems during spacecraft integration and test, or worse yet, during the mission.

As an example of the ambiguity of the assumed attitude control behavior, we note that it is not clear whether or not the attitude control subsystem will cause the spacecraft to adopt a controlled attitude as soon as it is provided with power, or what the controlled attitude might be. The description above could be interpreted as saying that upon being powered the attitude controller leaves the spacecraft uncontrolled, until it receives an attitude command. Alternatively, it could be argued that a transition to some (undefined) controlled attitude is implicit in the behavior description.

Either of the alternative behaviors for which we have just argued might be acceptable, or even required, depending on the mission. The problem is that the behavior description, as it stands, does not tell us which behavior should be assumed. A command subsystem designed under the assumption that the attitude controller automatically enforces a controlled attitude could inadvertently induce a catastrophic spacecraft state if interfaced with an attitude controller designed under the assumption that it should not bring the attitude under control until it has been told what attitude it needs to maintain. The sooner we can uncover such fundamental incompatibilities in assumed behavior, the easier (and cheaper) it is likely to be to modify the design of the subsystems concerned.

In the next few sections we explore how CSP process models can be used to provide less ambiguous descriptions of assumed subsystem behavior. As part of this exploration, in sec. 5.4.2 we revisit the attitude controller example discussed here, and illustrate how it can
be recast into a process model. In later sections we look at how subsystem process models can be used to examine the system-level implications of an assumed subsystem behavior, and catch incompatible assumptions early in the spacecraft design process.

5.2 Approach

The essence of our approach is to develop a CSP model of a spacecraft system that is structurally similar to a system block diagram (fig. 5.2). That is, we model subsystem blocks as processes that describe the abstract behavior of the subsystem, and model the lines that specify interfaces between subsystems as CSP channels. We represent specific interactions between subsystems as events associated with a specific channel. The parallel composition of subsystem models synchronizing on common channels produces a spacecraft system model, the behavior of which is a result of the interactions between the subsystems. A complete description of the subsystem interactions also requires auxiliary channels to capture dependencies between subsystems that are not usually shown in block diagrams, as well as subsystem responses to environmental changes and faults.

The process model of a subsystem defines the events in which the subsystem is able to participate in each subsystem state, and the response the subsystem has to a given event. In contrast to resource-oriented spacecraft models [27,28], we construct our subsystem models under the assumption that the continuous dynamics and closed-loop controllers embodied by the subsystem operate correctly. We instead focus on the exchanges of commands, telemetry, and other information between subsystems, and the response of each subsystem to changes in the state of other subsystems and of the environment. Although we model some aspects of resource consumption, our resource models are abstract in nature, and are used to define how the subsystems respond to qualitative changes in resource levels. Abstracting from the internal details of the subsystems allows systems engineers to focus on determining whether or not, under the assumption that the internal dynamics and control of each subsystem has been correctly designed, the interactions between subsystems lead to the desired spacecraft system behavior. The resulting system models are complementary to, rather than a replacement for, models produced using more traditional modeling approaches.
5.3 Subsystem Events

Since our approach involves modeling a subsystem as a CSP process built from sequences of events, it is worthwhile to examine the kinds of events which might be contained in a subsystem process model before embarking on the construction of such a model. We consider three broad categories of events, each representing a different aspect of subsystem behavior. These categories of events are: explicit interface events, implicit interface events, and specification events. For each category we provide example-driven guidelines for constructing the channels and associated types used to define the events found in that category. We follow the type declaration conventions used by CSP_M [48]. The guidelines are summarized in a table at the end of this section.

5.3.1 Explicit Interfaces

Explicit interfaces are the interfaces which are typically defined in spacecraft specification documents, and which appear in system block diagrams as lines connecting different subsystem boxes. A good example of a specification for the explicit interfaces of a subsystem, adapted from the ACE Spacecraft Design Specification [22], appears in fig. 5.3.

**Definition 32** (Explicit Interface). An *explicit interface* is a deliberately constructed information or power interface between subsystems, or between a subsystem and the environment.
<table>
<thead>
<tr>
<th>Power Subsystem Signal</th>
<th>Source or Destination</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main Bus Power</td>
<td>to fused loads</td>
<td>Power</td>
</tr>
<tr>
<td>Pyro Power</td>
<td>to solar array pyros</td>
<td>Power</td>
</tr>
<tr>
<td>Data Commands</td>
<td>from C&amp;DH</td>
<td>Command</td>
</tr>
<tr>
<td>Remote Relay Commands</td>
<td>from C&amp;DH</td>
<td>Command</td>
</tr>
<tr>
<td>LVS Main Bus Threshold</td>
<td>to C&amp;DH</td>
<td>Discrete Signal</td>
</tr>
<tr>
<td>LVS Battery Threshold #1</td>
<td>to C&amp;DH</td>
<td>Discrete Signal</td>
</tr>
<tr>
<td>LVS Battery Threshold #2</td>
<td>to C&amp;DH</td>
<td>Discrete Signal</td>
</tr>
<tr>
<td>Analog Voltage Telemetry</td>
<td>to C&amp;DH</td>
<td>Telemetry Stream</td>
</tr>
</tbody>
</table>

Fig. 5.3: Power subsystem interfaces for the ACE spacecraft.

In CSP, we represent an explicit interface as a channel. We adopt a convention of naming each channel for the kind of explicit interface it represents, extended where necessary with other identifying information to form a unique name. For example, a bus carrying commands might be named `cmdbus`, while a pair of unregulated 28-Volt power buses might be given the names `unreg_28V_powerbus_A` and `unreg_28V_powerbus_B`. This convention approximates existing practices for labeling interfaces in specifications and block diagrams.

In addition to a name, each interface channel has a type which defines the set of events associated with the interface. In practice, the type of each interface in a spacecraft model is necessarily mission-specific, and explicating these types can be considered part of the specification process. However, while the details vary across missions, the types associated with a particular kind of interface tend to have similar structures. We consider five kinds of interfaces commonly found in spacecraft block diagrams and specification documents: discrete signals, command interfaces, telemetry interfaces, data buses, and power buses.

**Discrete Signals**

Discrete signals are dedicated interfaces used to sense or manipulate discrete state information, as exemplified by the excerpts from the *ACE Spacecraft Design Specification* [22] shown in fig. 5.4. The type associated with a discrete signal is correspondingly simple. A collection of atomic symbols representing the possible discrete values will usually suffice.
6.3.1.1 Low Voltage Sense (LVS)

The power subsystem shall detect one or more of the low voltage conditions below and provide three separate low voltage sense signals to the C&DH component:

- Main Bus LVS indicating a bus voltage $< 26\text{V}$.
- Battery LVS threshold #1 indicating a battery bus voltage of $< 19.8\text{V}$.
- Battery LVS threshold #2 indicating a battery bus voltage of $< 18.9\text{V}$.

Fig. 5.4: Examples of discrete signals on the ACE spacecraft.

Example 5.3.1.1. The type for a discrete signal controlling a switch might be defined as a pair of atomic symbols representing the two possible switch states:

```
datatype OnOff = on | off

channel switch_discrete : OnOff
```

Example 5.3.1.2. The low voltage sense discrete signals mentioned in fig. 5.4 might be represented as:

```
datatype LVS = main_bus_lvs | battery_lvs_1 | battery_lvs_2

channel lvs_signal : LVS
```

Commands

Command interfaces carry discrete messages that request a change in state or behavior. The quantity and complexity of these messages varies depending on the complexity of the subsystems involved. Figure 5.5 shows an example of a definition of a command interface, excerpted from the *ACE Spacecraft Design Specification* [22].
6.2.10 Power Switching Component and Ordnance Component

... C&DH subsystem shall control the power switching or ordnance component to:

- Based on uplink relay commands:
  - Turn off any instrument or non-critical spacecraft component.
  - Turn on any instrument or non-critical spacecraft component.
  - Select thrusters.
  - Arm thrusters.
  - Remove instrument covers.
  - Deploy magnetometer boom.

Fig. 5.5: ACE power subsystem commands.

In some situations, the type associated with a command interface may be as simple as that for a discrete signal interface. However, in most cases it is convenient to define more complex, compound types to represent parameterized commands, instead of explicitly listing each individual command in the type declaration. This has the added advantage of allowing related commands to be grouped, making them easier to identify.

Example 5.3.1.3. The type $EPSCmd$ represents the commands that can be sent to a simple spacecraft electrical power system (EPS). The EPS responds to two different kinds of commands: commands to activate pyrotechnic deployment devices, denoted by the tag $pyro$, and commands to switch subsystems on or off, denoted by the tag $sw$. The type is

$$
\text{datatype } EPSCmd = sw.\text{Subsystem}.\text{OnOff} \mid pyro.\{\text{antenna.1, antenna.2}\}
$$

where $\text{Subsystem}$ is a set of subsystem names, and $\text{OnOff}$ is the set $\{\text{on, off}\}$.

The type declaration defines switch commands as compound values consisting of the tag $sw$, followed by two symbols drawn from the sets $\text{Subsystem}$ and $\text{OnOff}$, and representing command parameters. Thus, for example, the value $sw.\text{adcs.on}$ represents a command intended to power on the attitude determination and control subsystem. Pyrotechnic commands are similarly parameterized by the deployment they are intended to trigger.
A command interface to the EPS can be defined as a channel of the \textit{EPSCmd} type:

\begin{verbatim}
channel eps\_cmd : EPSCmd
\end{verbatim}

The events associated with this channel, such as \texttt{eps\_cmd.pyro.antenna.1} and \texttt{eps\_cmd.sw.adcs.on}, indicate the occurrence of different EPS commands.

\section*{Telemetry}

Telemetry interfaces carry data that provides information on the state of a subsystem. The telemetry data may be discrete messages that are sent when triggered by specific changes in subsystem state, or it may be a stream of information that is continually transmitted while the telemetry interface is active. Figure 5.6, excerpted from the \textit{WIRE System Requirements} [24], provides an example of the sort of information carried in a telemetry interface.

Simple telemetry data messages can be represented as atomic symbols. More complex data messages might require compound values, but still fit easily within the event-based style of interactions upon which CSP focuses. However, mapping telemetry streams onto a type requires more careful consideration of how the streams will actually be modeled.

In the sections that follow we avoid modeling the individual data items in a telemetry stream directly, but instead model the stream as a state-bearing process that is an abstraction of the underlying stream of data. The state of the stream at any given time is a symbol representing a defined range of underlying concrete telemetry values, and transitions in the stream state represent qualitative changes in the concrete telemetry values (fig. 5.7). Since telemetry streams may not always be actively streaming data, the set of possible telemetry stream states should also include an “inactive” value.

\textbf{Example 5.3.1.4.} The set of states for a stream of attitude telemetry can be defined as the union of the \textit{Attitude} type (defined in the previous chapter), which consists of symbolic names representing different concrete attitude angles and rates, and the \textit{inactive} state.

\begin{verbatim}
nametype AttitudeTlm = Attitude \cup \{ inactive \}
\end{verbatim}
CDH.REQ.GEN.25 SOLAR ARRAY DEPLOYMENT TELEMETRY
The following telemetry from the solar array deployment mechanisms shall be available in the lowest-rate telemetry stream:

- Latch power status (actuators powered or not powered) for all latches
- Latch status (released or not released) for all latches
- Array fully deployed (yes or no) for both panels
- Angle of array rotation (analog signal)

Fig. 5.6: Telemetry for the WIRE array deployment mechanism.

Fig. 5.7: Abstract states Attitude 1 and Attitude 2 represent qualitatively different attitudes.

The interface to a telemetry stream state consists of the same kind of set, get, and state transition events as the AssignableState process defined in the previous chapter. For convenience, we define a datatype that captures this interface.

**Definition 33** (State Interface Datatype).

\[
\text{datatype } \text{StateIF} = \text{setval} | \text{getval} | \text{trans}
\]

In practice, only the process model for the subsystem producing the telemetry stream will use the setval portion of the StateIF interface, since the producing subsystem is the only entity that should reasonably be able to alter the telemetry stream. Process models for
subsystems that receive the telemetry stream use the \textit{getval} and \textit{trans} parts of the interface to find the current state of the stream, or to detect transitions in the stream state.

The type representing a telemetry stream consists of compound values built from the state interface datatype and a type containing values for stream states. When the telemetry stream is defined as part of a larger type involving other telemetry data, the type should also include a tag identifying the stream.

\textbf{Example 5.3.1.5.} The type $ADCSTlm$ represents telemetry that may be sent from an attitude determination and control system. The type includes values for two discrete messages, one of which is used to signal receipt of an attitude command, and the other to signal violation of an attitude constraint. Also included within the type is a set of compound values representing a stream of attitude data, prefixed with the identifying tag $att\_tlm\_stream$.

\begin{verbatim}
    datatype ADCSTlm = att\_violation
                     | att\_cmd\_ack
                     | att\_tlm\_stream.StateIF.AttitudeTlm
\end{verbatim}

The $adcs\_tlm$ channel models a telemetry interface to the ADCS:

\begin{verbatim}
    channel adcs\_tlm : ADCSTlm
\end{verbatim}

The event $adcs\_tlm.att\_violation$ represents transmission of a message from the ADCS indicating an attitude constraint violation of some kind.

The event $adcs\_tlm.att\_tlm\_stream.trans.earth\_limb\_scan$ indicates that spacecraft attitude reported by the stream of attitude telemetry data from the ADCS has made a qualitative transition to the $earth\_limb\_scan$ attitude.

\textbf{Data Buses}

A data bus aggregates multiple subsystem-to-subsystem interfaces into a single shared interface. Although it is feasible to model a data bus as a number of separate channels, each representing an individual subsystem-to-subsystem interface provided by the bus, that
approach is less than ideal if we wish to preserve as much as possible of the structure of a block diagram within the corresponding process model. Instead, we model a data bus as a single channel which has a structured type that captures the individual subsystem interfaces within the bus.

In the case of command interfaces aggregated onto a bus, it is important to identify the intended recipient of each command. We therefore define the tags that delineate the structure of a command bus type such that they include recipient information.

**Example 5.3.1.6.** The $SubsysCmd$ type represents an aggregation of several different command interfaces onto a single bus. The tags $adcs\_cmd$, $eps\_cmd$, $pl\_cmd$, and $dl\_cmd$ identify the different command recipients.

\[
\text{datatype} \ SubsysCmd = adcs\_cmd.\text{AttitudeCommand} \\
| eps\_cmd.\text{EPSCmd} \\
| pl\_cmd.\text{PayloadCmd} \\
| dl\_cmd.\text{DownlinkMsg}
\]

The recipient tags are prefixed onto already-defined types representing the available commands for different subsystems. For example, the $eps\_cmd$ tag, which identifies commands intended for the EPS, is prefixed to the $EPSCmd$ type defined in example 5.3.1.3.

Note that in the preceding discussion we have assumed that the source of a command is not relevant to the execution of the command. In situations where this is not a valid assumption, it may be necessary to add a source tag to each type of command, in addition to the recipient tags. Alternatively, the command source could be included as a command parameter within a compound type representing a parameterized command.

In contrast to command interfaces, it is typically the source of telemetry data, rather than the recipient, that is important. Thus, the tags that we use to structure a telemetry bus type identify the source of the telemetry rather than the recipient.
Example 5.3.1.7. The $SubsysTlm$ type represents an aggregation of several different telemetry interfaces onto a single bus. The tags $adcs\_tlm$ and $pl\_tlm$ identify the different telemetry sources.

$$\text{datatype } SubsysTlm = adcs\_tlm.ADCSTlm$$

$$| pl\_tlm.PayloadTlm$$

In general, a spacecraft data bus will carry both commands and telemetry between the subsystems it connects.

Example 5.3.1.8. The $systembus$ channel represents a spacecraft data bus (fig. 5.8). The $SysBus$ type combines both the previously defined command and telemetry bus types into a single type that defines the set of messages associated with the $systembus$ channel.

$$\text{name type } SysBus = SubsysCmd \cup SubsysTlm$$

$$\text{channel } systembus : SysBus$$

Within this framework, the event $systembus.\text{eps}\_cmd.\text{sw}.\text{adcs}.\text{on}$ represents a command, delivered via the $systembus$ channel, to switch on power to the attitude determination and control subsystem. Similarly, a qualitative transition to the $earth\_limb\_scan$ attitude is indicated by the event $systembus.adcs\_tlm.att\_tlm.stream.trans.earth\_limb\_scan$.  

![Fig. 5.8: Connectivity of the $systembus$ channel.](image-url)
Power

Power interfaces, as the name suggests, carry electrical power into a subsystem. As with telemetry streams, mapping power interfaces onto CSP events requires some consideration of how the interface is to be modeled. In modeling how subsystems can interact through a power interface we are interested in two things:

1. External influences on a given subsystem, i.e. whether or not power is being supplied to the subsystem through the power interface.

2. The impact of a given subsystem on other subsystems, i.e. how much power a subsystem is consuming through the power interface.

The first of the two kinds of power interaction can be represented using a compound value consisting of a subsystem identifier, and a symbol indicating whether the identified subsystem is being provided with power (on) or not (off). Events built using these values indicate that an external subsystem has elected to provide power to (or alternatively to withdraw power from) the identified subsystem.

The second kind of interaction is most easily modeled in terms of changes in the level of power being consumed by a particular subsystem. It thus lends itself to representation using a compound value composed of a subsystem identifier, and a numerical value indicating the quantity of change in consumed power for the identified subsystem. In section 5.4.4 we describe how to build a process model of a power source which responds to events representing changes in subsystem power consumption.

Example 5.3.1.9. The type Power represents the possible interactions on a power interface. Values prefixed with the symbol load_switch indicate whether or not a subsystem is being supplied power. Values prefixed with the symbol load_delta indicate changes in power consumed by a subsystem. In this example the allowable magnitudes of a change in subsystem power consumption are restricted to a finite range of integer values, which helps prevent the state-space of models built using this type from growing too large.
datatype \texttt{Power} = \texttt{load\_switch.Subsystem.\texttt{OnOff}}
\hspace{1cm} \mid \texttt{load\_delta.Subsystem.}\{-20 \ldots 20\}

The channel \texttt{power} has the type \texttt{Power}:

\begin{verbatim}
channel \texttt{power} : \texttt{Power}
\end{verbatim}

Examples of events involving this channel include: \texttt{power.load\_switch.cdh.on}, which indicates that power is now being supplied to the command and data-handling system, and \texttt{power.load\_delta.cdh.5}, which indicates that the amount of power being consumed by the command and data-handling system has increased by 5 units. The significance of these two kinds of power events will become clearer in later sections, when we examine how to construct abstract process models of a spacecraft electrical power system.

\section*{5.3.2 Implicit Interfaces}

Implicit interfaces represent subsystem behavior dependencies that are implicit in the design of the spacecraft. They are typically not explicitly shown on system block diagrams. A simple example of an implicit interface is the dependency of the output of a solar-array-based spacecraft electrical power system on the attitude of the spacecraft, which is influenced by the attitude determination and control system (fig. 5.9, excerpted from the \textit{WIRE System Requirements} [24]). The output of the electrical power system is similarly dependent on the eclipse state of the spacecraft, which is influenced by the spacecraft orbit and environment.

\begin{center}
\textbf{ACS.REQ.ACQ.2 SAFEHOLD SUN POINTING ACCURACY}
\begin{quote}
\textit{After sun acquisition, all safehold modes shall maintain the spacecraft y-axis within 30 degrees of the sunline. Rationale:} This angle is required to maintain positive power margin.
\end{quote}
\end{center}

\begin{verbatim}
Fig. 5.9: Example of an implicit interface on the WIRE spacecraft.
\end{verbatim}
**Definition 34** (Implicit Interface). An *implicit interface* is a point of interaction which results when the behavior of a subsystem depends upon a physical state that is under the control of another subsystem, or of the environment.

We adopt a convention of naming the channel representing an implicit interface for the physical state with which the interface is associated. For example, an implicit interface resulting from a dependency on spacecraft attitude might be represented by a channel named *attitude* (fig. 5.10).

Because implicit interfaces do not carry structured information, the types associated with such interfaces can be relatively simple. The underlying physical state that produces the implicit interface can be abstracted into a collection of atomic symbols that represent qualitatively different concrete state values.

**Example 5.3.2.1.** The attitude of a spacecraft might be represented by the datatype

```
datatype Attitude = uncontrolled | earth_limb_scan | sun_pointing
```

which in this case consists of the same set of abstract attitude states used in the examples of the previous chapter. In constructing a channel to represent the implicit interface that results from the spacecraft attitude state, we make use of the state interface type introduced in definition 33 to represent the usual operations on states:

```
channel attitude : StateIF.Attitude
```

---

Fig. 5.10: Explicit power interface, and implicit attitude interface.
5.3.3 Specification Events

Specification events are abstractions that provide a way to express and manipulate certain aspects of subsystem behavior that are important for defining or verifying the interactions between subsystems, but that don’t directly appear in any subsystem interface. For example, we may wish to specify that some internal error shouldn’t ever occur within a subsystem, without having to model the internal details of how that error might arise. Or, as in the excerpt from the *WIRE System Requirements* [24] shown in fig. 5.11, we may wish to define how the observable behavior of a subsystem changes in the event of an internal hardware failure, again without having to provide details of the exact failure mechanism.

**Definition 35** (Specification Event). A *specification event* is an abstract representation of some aspect of the behavior of a subsystem that is not directly observable at the subsystem interface.

It is difficult to provide specific guidelines for the definition of specification events, since the kind of events that we might wish to represent are likely to vary depending on the mission under consideration, and the properties of the spacecraft design that we want to verify. To the extent that it makes sense to establish guidelines, those guidelines are broadly applicable to the development of any kind of CSP process model. For example, it is generally good practice to collect related specification events into a common datatype associated with a single channel, simply because this practice makes it easier to manage the related events when composing individual processes to form large models. Similarly, it

### ACS.REQ.FDH.2 SUN / EARTH CONSTRAINT CHECKING

*The Failure Detection and Handling System [of the Attitude Control Subsystem] shall be able to autonomously detect a violation of the Sun Avoidance Constraint or the Earth Avoidance Constraint while operating in any of the SCS controlled modes. Upon detection of a sustained violation, the FDH system shall autonomously transfer control to the next lower operating mode.*

Fig. 5.11: Example of abstract specification events.
is helpful to define the symbols within the datatype that represents a set of specification events using descriptive names, since this practice makes the resulting specifications more easily comprehensible.

**Example 5.3.3.1.** The ADCSFaultEvent type and its associated adcs_fault channel provide an abstract representation of four different classes of internal faults that might occur within an attitude determination and control subsystem. These abstract events represent the output of some kind of fault detection and classification mechanism within the ADCS.

```
datatype ADCSFaultEvent = hardware_anomaly
    | hardware_failure
    | good_star_check_failure
    | sun_earth_constraintViolation

channel adcs_fault : ADCSFaultEvent
```

### 5.3.4 Summary

Table 5.1 provides a summary of the guidelines for modeling spacecraft interfaces that we have described in this section.

<table>
<thead>
<tr>
<th>Domain concept</th>
<th>CSP concept</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit Interface</td>
<td>Channel named for interface</td>
</tr>
<tr>
<td>Discrete signals</td>
<td>Atomic symbols</td>
</tr>
<tr>
<td>Simple commands</td>
<td>Atomic symbols</td>
</tr>
<tr>
<td>Parameterized commands</td>
<td>Compound values</td>
</tr>
<tr>
<td>Telemetry streams</td>
<td>Data states &amp; inactive state</td>
</tr>
<tr>
<td>Databuses</td>
<td>Source or recipient tagged messages</td>
</tr>
<tr>
<td>Power interfaces</td>
<td>Switch and delta events</td>
</tr>
<tr>
<td>Implicit Interface</td>
<td>Channel named for physical state</td>
</tr>
<tr>
<td>Specification Event</td>
<td>Abstract states</td>
</tr>
</tbody>
</table>

→ Common datatype for related events
5.4 Subsystem Behavior

A subsystem behavior model describes how a spacecraft subsystem reacts to commands and other inputs, under which conditions it generates particular types of outputs, how its power consumption varies over different operating modes, and how it responds to internal faults. The way in which a spacecraft system is divided into its constituent subsystems is fairly well established [19, 29, 30], and although the specific behavior required of a particular subsystem is dependent on the mission the spacecraft is designed to perform, the role of each subsystem is broadly consistent across different missions. As a result, for each kind of subsystem we can identify a general approach for modeling the subsystem behavior, thereby providing a starting point for the development of mission-specific subsystem models.

We begin by providing definitions for several generic behaviors which are found in more than one subsystem, and that can be easily abstracted into reusable process descriptions. We then use the attitude determination and control subsystem as the focus of a series of examples which introduce modeling approaches for several different aspects of subsystem behavior, many of which apply to other subsystems as well. We next consider the command and data-handling subsystem and electrical power subsystem. These two subsystems play a key role in defining system behavior, since they typically touch all of the other subsystems, and both require somewhat different approaches to modeling than the attitude determination and control subsystem. Lastly, we look at the other spacecraft subsystems, although in less detail than the attitude, command, and power subsystems, since the modeling approaches for the remaining subsystems are essentially the same as those introduced in considering the first three subsystems.

5.4.1 Generic Behavior Models

Two kinds of behavior which seem to crop up regularly when trying to model spacecraft subsystems are streams of telemetry data that report on some state controlled by a subsystem, and changes in subsystem power consumption that are dependent on the mode in which the subsystem is operating. Since we need to implement these behaviors over and over again, it is convenient to capture them as reusable process models.
As mentioned in section 5.3.1, our approach to modeling streams of telemetry data is based on abstracting the telemetry stream into a state representing the underlying concrete telemetry values. Modeling telemetry streams in this way avoids the intractably large state-spaces that could result from mapping each individual message in the stream to an event, while also focusing attention on that aspect of the streaming telemetry which actually impacts the behavior of the subsystems reading data from the stream.

The generic telemetry stream process is constructed under the assumption that the data in the stream represents some kind of subsystem state information. Transitions in the subsystem state produce corresponding transitions in the telemetry data state. The telemetry stream itself may be either inactive, in which case telemetry data is unavailable, or active, in which case telemetry data is available to consumers of the stream. Signals on a control channel allow the stream to be switched to and from its inactive state.

**Definition 36** (State Telemetry Stream). A *state telemetry stream* is a process

\[
\text{StateTelemetryStream}(\text{mode, state\_get, state\_trans, set, get, trans, f}) = \\
\quad \text{let} \\
\quad \quad \text{Inactive} = \text{state\_trans}?x \rightarrow \text{Inactive} \\
\quad \quad \quad \quad \Box \\
\quad \quad \quad \text{mode\_off} \rightarrow \text{Inactive} \\
\quad \quad \quad \quad \Box \\
\quad \quad \quad \text{mode\_on} \rightarrow \text{Setup} \\
\quad \quad \text{Setup} = \text{state\_trans}?x \rightarrow \text{set}\!f(x) \rightarrow \text{Active} \\
\quad \quad \quad \quad \Box \\
\quad \quad \quad \text{state\_get}?x \rightarrow \text{set}\!f(x) \rightarrow \text{Active} \\
\quad \quad \quad \quad \Box \\
\quad \quad \quad \text{mode\_off} \rightarrow \text{set.inactive} \rightarrow \text{Inactive} \\
\quad \quad \quad \quad \Box \\
\quad \quad \quad \text{mode\_on} \rightarrow \text{Setup}
\]
\[\text{Active} = \text{state\_trans}\?x \rightarrow \text{set}!f(x) \rightarrow \text{Active}\]

\[
\square
\]

\[\text{mode\_off} \rightarrow \text{set\_inactive} \rightarrow \text{Inactive}\]

\[
\square
\]

\[\text{mode\_on} \rightarrow \text{Active}\]

\[\text{StreamData} = \text{AssignableState}(\text{set}, \text{get}, \text{trans}, \text{inactive})\]

within

\[
(\text{Inactive} [[\{\text{set}\}]] \text{StreamData}) \setminus \{\text{set}\}
\]

where

- \(\text{mode}\) is a control channel carrying values \(\text{on}\) and \(\text{off}\).

- \(\text{state\_get}\) and \(\text{state\_trans}\) are channels which provide the interface between the telemetry stream and the subsystem state the stream represents.

- \(\text{set}, \text{get},\) and \(\text{trans}\) are channels which provide an interface to the current state of the telemetry stream data.

- \(f\) is a function which translates subsystem states into telemetry data states (typically the identity function).

- \(\text{inactive}\) is a symbol denoting an inactive telemetry stream.

- \(\text{StreamData}\) is an assignable state process representing the current value of the telemetry stream data.

- \(\text{Inactive}\) is a process representing the inactive state of the telemetry stream, in which changes in the underlying subsystem state are discarded.

- \(\text{Setup}\) is a process representing the transition from an inactive telemetry stream to an active stream, in which the value of the telemetry data state is changed to represent the underlying subsystem state.
• *Active* is a process representing an active telemetry stream, in which transitions in the underlying subsystem state produce changes in the telemetry data state.

The telemetry stream is initially inactive, and has a data state *inactive*. Whenever the telemetry stream is switched back to the inactive state the stream data state is reset to *inactive*.

The generic subsystem mode power process translates transitions in the operating mode of a subsystem into transitions in the quantity of power consumed by the subsystem. The power delta events produced by the mode power process are intended to be consumed by an electrical power system process model of the sort described later in this section.

**Definition 37** (Subsystem Mode Power). A *Subsystem mode power* behavior is a parameterized process

\[
\text{SubsysModePower}(\text{initmode}, \text{modetrans}, \text{power\_delta}, f_{\text{ModePower}}) =
\]

\[
\text{let}
\]

\[
f_{\text{PowerDelta}}(m, m') = f_{\text{ModePower}}(m') - f_{\text{ModePower}}(m)
\]

\[
P_{\text{Mode}}(m) =
\]

\[
\text{modetrans.begin?m' } \rightarrow \text{ if } m \neq m'
\]

\[
\text{then } \text{power\_delta!}f_{\text{PowerDelta}}(m, m')
\]

\[
\rightarrow \text{modetrans.end.m' } \rightarrow \text{PMode}(m')
\]

\[
\text{else } \text{modetrans.end.m } \rightarrow \text{PMode}(m)
\]

\[
\text{within}
\]

\[
P_{\text{Mode}}(\text{initmode})
\]

where

• *initmode* is the mode in which the subsystem is initially assumed to be operating.

• *modetrans* is a channel through which the beginning and end of subsystem mode transitions is signaled.
- $power\_delta$ is a channel through which quantitative changes in subsystem power consumption are signaled.

- $f_{ModePower}$ is a function from subsystem mode names to power consumption values.

- $f_{PowerDelta}$ is an auxiliary function which computes the difference in power consumption between two subsystem modes.

- $PMode$ is a process which receives $modetrans$ events, and produces $power\_delta$ events if a change in the subsystem operating mode has occurred.

Both of these generic behaviors are included in the library of CSP constructs which appears in appendix A. We present examples of the use of both of these generic behaviors later in this section. Example 5.4.2.2 introduces the use of State Telemetry Streams, and example 5.4.2.4 introduces the use of Subsystem Mode Power processes.

It's worth noting that these two generic behaviors should not be considered the only generic behaviors, although they are the only ones that we have identified so far. Other common kinds of behavior may come to light as we gain more experience with modeling spacecraft subsystems.

### 5.4.2 Attitude Determination and Control

The attitude determination and control subsystem (ADCS) is responsible for orienting the spacecraft, and for changing the spacecraft orientation (or attitude) in response to commands. The ADCS may also provide information on the spacecraft attitude and rates of change in attitude to other subsystems.

At any given time, the spacecraft has a nominal attitude which it should ideally maintain. For example, the nominal attitude of a geostationary communications satellite through most of its mission is an earth-pointing attitude. Similarly, the nominal attitude of a scientific spacecraft is the attitude which keeps its instruments pointed at scientific targets of interest, an attitude which may vary depending upon which targets the spacecraft has been commanded to study.
Internally, a typical ADCS uses some kind of feedback-based control to manipulate various actuators which provide the torques necessary to change or maintain the attitude. The type of sensors, actuators, control laws, and attitude determination algorithms that are selected for the ADCS depend on the kind of attitudes the spacecraft is expected to attain, and the accuracy with which it is required to maintain a particular attitude [8]. However, in constructing a model of spacecraft subsystem interactions we’re not really concerned with the details of exactly how the ADCS achieves any required changes in spacecraft attitude state, but rather what changes in attitude state take place, and what actions on the part of other subsystems trigger those changes.

A Simple Model

A simple model of the attitude control portion of an ADCS can be constructed by defining a process in which commands are mapped to attitude state changes when the controller is active, and the attitude state is left uncontrolled when the controller is not active. The underlying assumption of this model is that the design of the attitude controller is sufficient to meet all of the spacecraft attitude requirements, i.e. the controller can adequately cancel any disturbance torque that might be encountered during the spacecraft mission, can achieve any attitude it might be commanded into, and meets all performance requirements. As the excerpt from the *MGS Spacecraft Requirements* [23] shown in fig. 5.12 illustrates, exactly this assumption can be found in existing spacecraft requirements documents.

Example 5.4.2.1. Consider a simple ADCS for a science spacecraft. The development of a process model for this subsystem begins with the creation of datatypes which define the

---

**5.4 ATTITUDE AND ARTICULATION CONTROL SUBSYSTEM**

*The spacecraft bus shall have sufficient control authority to automatically maintain attitude orientation and stability of the spacecraft during all phases of the mission . . . The AACS shall also be capable of pointing the body mounted science instruments at arbitrary targets in the celestial frame.*

---

Fig. 5.12: Assumed behavior of the MGS attitude control system.
interface of the subsystems. As part of the mission design process, we have presumably defined a small set of qualitatively different attitude states which the spacecraft may attain at different points during its mission. These states form the basis for the \textit{Attitude} datatype:

\begin{verbatim}
datatype Attitude = uncontrolled
                   | rate_nulled
                   | sun_safe
                   | earth_pointing
                   | science_target.{1..5}
\end{verbatim}

We assume that it should not be possible to command the spacecraft into an \textit{uncontrolled} attitude, and define a set of commands which allow all other attitudes to be commanded. For the purposes of this example, we assume that the only commands the ADCS responds to are attitude commands.

\begin{verbatim}
nametype CommandableAttitude = Attitude \{uncontrolled\}
datatype AttitudeCmd = set_attitude.CommandableAttitude
datatype ADCScmd = attcmd.AttitudeCmd
\end{verbatim}

The interface of the ADCS model (fig. 5.13) consists of explicit interface channels for commands and power, and an implicit interface channel which carries attitude state information:

\begin{verbatim}
channel cmd : ADCScmd
channel power : Power
channel attitude : StateIF.Attitude
\end{verbatim}

where the \textit{Power} datatype is as defined in the example which appears in section 5.3.1, and we assume that the set \textit{Subsystem} includes a symbol \textit{adcs}. 
The assumed behavior of the simple ADCS is similar to that described in the motivating example from section 5.1:

- The ADCS is initially unpowered.
- When the ADCS is not powered, the spacecraft attitude is *uncontrolled*, and commands have no effect.
- When power is supplied to the ADCS, the subsystem transitions into a mode in which it is able to make spacecraft attitude changes in response to received commands, every attitude command results in the commanded attitude being achieved.
- Removing power from the ADCS causes the attitude to again become *uncontrolled*, and the ADCS to ignore commands.

The \( \text{SimpleADCS}_1 \) process model captures this behavior:

\[
\text{SimpleADCS}_1 = \\
\quad \text{let} \\
\quad \quad \text{Controller}(\text{off}) = \\
\quad \quad \quad \text{cmd}.\text{attcmd}.\text{set}\_\text{attitude}\?a \rightarrow \text{Controller}(\text{off}) \\
\quad \quad \quad \square \\
\quad \quad \text{power}.\text{load}\_\text{switch}\.\text{adcs}.\text{on} \rightarrow \text{Controller}(\text{on})
\]
Controller(on) =
    cmd.attcmd.set_attitude?a → attitude.setval!a → Controller(on)
\[\square\]
    power.load_switch.adcs.off → attitude.setval!uncontrolled → Controller(off)
AttitudeState(init) =
    AssignableState(attitude.setval, attitude.getval, attitude.trans, init)
within
        (Controller(off)
        \[\{\text{attitude.setval}\}\])
AttitudeState(uncontrolled) \{\text{attitude.setval}\}

The SimpleADCS₁ process encapsulates both the attitude state, and the controller which effects changes to that state. Events with an attitude.setval prefix are hidden, which prevents processes other than the controller from changing the attitude state. Hiding the setval events in this way effectively encodes the assumption that ADCS can maintain complete control of the attitude state, regardless of the actions of other subsystems. We leave the other attitude state events available, since other subsystems should be able to sense the attitude state (although it is more likely that other subsystems will obtain their attitude data via the ADCS instead of directly sensing it themselves).

Although attitude determination is a necessary prerequisite to attitude control, the SimpleADCS₁ model completely encapsulates that aspect of the function of the ADCS, since attitude information is not propagated outside the black-box of the attitude controller. Some missions, though, require that the ADCS provide attitude information to other subsystems, for example to enable scientific measurements to be tagged with the attitude in which they were collected. The communication of attitude data to other subsystems can be orchestrated in several different ways, each of which has different implications for the subsystems to which the ADCS interfaces. For example, we might design the ADCS to produce discrete attitude data messages in response to requests from other subsystems (we examine an example of this style of data production in section 5.4.6). Alternatively, instead of requiring proactive
requests from other subsystems, the ADCS may be designed to transmit a continuous stream of attitude state data.

**Example 5.4.2.2.** The $SimpleADCS_1$ process can be extended to provide attitude data to other subsystems by adding to it a generic $StateTelemetryStream$ behavior (definition 36), which models the production of a continuous stream of attitude data.

Following the guidelines laid out in section 5.3.1, we establish a telemetry datatype consisting of the union of the $Attitude$ datatype and the $inactive$ state, and a telemetry channel which provides access to telemetry values via a State Interface:

$$
\text{datatype } AttitudeTlm = Attitude \cup \{ \text{inactive} \}
$$

$$
\text{channel } tlm : StateIF.AttitudeTlm
$$

The new ADCS process model, which we call $SimpleADCS_2$, is formed by composing $SimpleADCS_1$ with a $StateTelemetryStream$ process. The $SimpleADCS_2$ model (fig. 5.14) includes the implicit attitude interface and explicit cmd and power interfaces of the $SimpleADCS_1$ model, and adds an explicit telemetry interface, $tlm$. In constructing $SimpleADCS_2$, we assume that the state telemetry stream is always actively streaming data whenever the ADCS is powered, and therefore use $power.load\_switch.adcs$ events to activate and deactivate the stream. We further assume that the telemetry stream directly reports on the attitude state, and therefore use the identity function, $f_{ID}(x) = x$, as the telemetry stream processing function.

$$
SimpleADCS_2 =
(SimpleADCS_1
||[\{\text{power.load\_switch.adcs, attitude.getval, attitude.trans}\}]\]
StateTelemetryStream(power.load\_switch.adcs, attitude.getval,) \setminus \{tlm.setval\}
attitude.trans, tlm.setval, tlm.getval,

tlm.trans, f_{ID})
Synchronizing on $\text{tlm.getval}$ permits other subsystem processes to obtain the current state of the attitude telemetry stream, which will be either an attitude state value, or the inactive state. The continuous nature of the stream is reflected in the fact that other processes can synchronize on $\text{tlm.getval}$ at any time, without first having to send a request or otherwise trigger the delivery of attitude data. Events in $\{\text{tlm.trans}\}$ indicate changes in the state of the stream, and can be used to trigger corresponding changes in the behavior of any subsystem which is a consumer of the telemetry data. Events in $\{\text{tlm.setval}\}$ are hidden, to prevent other processes from modifying the telemetry stream state.

**More Complex Behavior**

The $\text{SimpleADCS}_1$ and $\text{SimpleADCS}_2$ models possess fairly straightforward behavior, and provide a good starting point for specifying an ADCS. In some cases, such simple models may be sufficient to capture all of the relevant ADCS behavior. However, many spacecraft designs incorporate an ADCS with more complex, mode-based behavior, including limitations on when certain attitude transitions can be made, diverse mode transition triggers, variations in acceptable inputs and outputs across different modes, and changes in subsystem power consumption depending on which sensors and actuators are assumed be operating in different modes. Figure 5.15, adapted from a diagram in the *EO-1 Preliminary Technology and Science Validation Report* [118], shows an example of the ADCS modes, and the possible transitions between them, for the EO-1 spacecraft.
The \textit{SimpleADCS}_1 process models an ADCS which effectively has two modes: powered, and unpowered (represented by the symbols \textit{on} and \textit{off}). Transitions between the two modes are triggered by \texttt{power.load\_switch.adcs} events. An ADCS model having a more complex mode-based behavior is built in the same style as \textit{SimpleADCS}_1, but with an increased the number of modes, and additional transition triggers beyond \texttt{power.load\_switch.adcs} events. The resulting process is similar in concept to a mode transition behavior defined in terms of the state transition systems described in section 4.2.2. However, for compactness we typically include simple mode-specific behaviors directly in the definition of the mode transition process, and reserve the use of mode constraints for capturing more complex mode-specific behavior.

\textbf{Example 5.4.2.3.} Consider an ADCS for a scientific spacecraft, with the following modes of behavior:

- An unpowered mode, in which the ADCS does nothing.
- A detumbling mode, in which the ADCS nulls the spacecraft attitude rates.
- A safehold mode, in which the ADCS achieves and maintains a safe and power-positive attitude.
- A science standby mode, in which the ADCS achieves and maintains a suitable preparatory attitude for the commencement of science operations.
• A science active mode, in which the ADCS responds to commands to adopt specific attitudes required for making scientific measurements.

The possible transitions between these different modes, and the kind of events that can trigger those transitions, are depicted in fig. 5.16.

To model the ADCS in CSP, we begin by defining a datatype which specifies the different ADCS modes. For convenience, we also define a subset of the ADCS mode datatype, consisting only of those modes which can be entered as the result of command. The occurrence of a mode transition is signaled by specification events on the channel \textit{modetrans}, which has type \textit{ModeTrans}.\textit{ADCSmode}. The values in the \textit{ModeTrans} type are used to indicate the beginning and end of a mode transition, which permits us to specify that certain events occur during the mode transition.

\begin{verbatim}
 datatype ADCSmode = unpowered | detumble | safehold | science_standby | science_active

 nametype CommandableADCSmode = ADCSmode \{unpowered\}

datatype ModeTrans = begin | end

 channel adcs_modetrans : ModeTrans.ADCSmode
\end{verbatim}

For the purposes of this example, we assume that the spacecraft attitude states are drawn the \textit{Attitude} datatype, which was defined in example 5.4.2.1. Changes in the attitude state are triggered by commands received via the ADCS command interface. We define two kinds of commands: mode commands, which tell the ADCS to transition to a particular mode, and attitude commands, which tell the ADCS to a specific attitude. Attitude commands are only applicable to science attitudes, since all other mission attitudes are produced as byproducts of a mode transition. The function \(f_{\text{ModeAtt}}\) defines the mapping from modes to attitude states.
Fig. 5.16: ADCS mode transitions for the model discussed in example 5.4.2.3.

name
type $\text{CommandableAttitude} = \{\text{science_target}\}$

datatype $\text{AttitudeCmd} = \text{set_attitude}.\text{CommandableAttitude}$

datatype $\text{ADCScmd} = \text{attcmd}.\text{AttitudeCmd} \ | \ \text{modecmd}.\text{CommandableADCSmode}$

channel $\text{cmd} : \text{ADCScmd}$

$f_{\text{ModeAtt}} : \text{ADCSmodes} \rightarrow \text{Attitude}$

$f_{\text{ModeAtt}} = \{(\text{unpowered}, \text{uncontrolled}),$

$(\text{detumble}, \text{rate_nulled}),$

$(\text{safehold}, \text{sun_safe}),$

$(\text{science_standby, earth_pointing}),$

$(\text{science_active, earth_pointing})\}$

To allow a more compact process model, we define some auxiliary functions that specify the sets of acceptable and unacceptable mode transition commands for each mode.

$\text{AllowedModeCmd}(\text{detumble}) = \{\text{safehold}\}$

$\text{AllowedModeCmd}(\text{safehold}) = \{\text{detumble, science_standby}\}$

$\text{AllowedModeCmd}(\text{science_standby}) = \{\text{safehold, science_active}\}$

$\text{AllowedModeCmd}(\text{science_active}) = \{\text{safehold, science_standby}\}$

$\text{DisallowedModeCmd}(m) = \text{CommandableADCSmodes} \setminus \text{AllowedModeCmd}(m)$
The ADCS telemetry interface is a mix of streaming attitude data, and a few simple, discrete messages related to the acceptance and rejection of mode commands. For simplicity, we assume that all of this telemetry is communicated through a single channel.

\[
\text{datatype } \text{ADCStlm} = \text{adcs\_ack\_cmd}\text{.CommandableADCSmode} \\
| \text{adcs\_reject\_cmd}\text{.CommandableADCSmode} \\
| \text{adcs\_stream}\text{.StateIF}\text{.AttitudeTlm}
\]

\[
\text{channel } tlm : \text{ADCStlm}
\]

where \text{AttitudeTlm} is defined in example 5.4.2.2.

The ADCS process model is the process \text{ComplexADCS}_1, defined below.

\[
\text{ComplexADCS}_1 = \text{let Controller}(unpowered) = \\
(\text{power.load\_switch.adcs.on} \\
\rightarrow \text{adcs\_modetrans.begin!detumble } \rightarrow \text{attitude.setval!rate\_nulled} \\
\rightarrow \text{adcs\_modetrans.end!detumble } \rightarrow \text{Controller(detumble)}) \\
\text{Controller}(m) = \\
(\text{power.load\_switch.adcs.on } \rightarrow \text{Controller}(m)) \\
(\text{power.load\_switch.adcs.off } \rightarrow \text{Controller}(unpowered)) \\
(\text{cmd.modecmd?m } \rightarrow \text{Controller}(unpowered))
\]
\((\text{cmd.modecmd}\,m') : \text{AllowedModeCmd}(m) \rightarrow \text{tlm!adcs_ack_cmd.m'}\)
\(\rightarrow \text{adcs_modetrans.begin!m'} \rightarrow \text{attitude.setval!f}_{\text{ModeAtt}}(m')\)
\(\rightarrow \text{adcs_modetrans.end!m'} \rightarrow \text{Controller}(m')\)

\(\square\)

\((\text{cmd.modecmd}\,m') : \text{DisallowedModeCmd}(m)\)
\(\rightarrow \text{tlm!adcs_reject_cmd.m'} \rightarrow \text{Controller}(m')\)

\(\text{ScienceTargeting} = \text{LiftF(cmd.attcmd.set\_attitude, attitude.setval, fID)}\)

\(\text{TargetingModeConstraint} =\)

\(\text{ModeConstraint}\{(\text{cmd.attcmd}, \text{adcs_modetrans.end}, \text{science_active, ADCSmode} \setminus \{\text{science_active}\})\}\)

\(\text{AttitudeState}(\text{init}) =\)

\(\text{AssignableState}(\text{attitude.setval, attitude.getval, attitude.trans, init})\)

within

\(((\text{Controller(unpowered)} \parallel \text{ScienceTargeting})\)
\((\text{cmd.attcmd, adcs_modetrans.end}) \parallel \text{TargetingModeConstraint})\)
\((\text{attitude.setval}) \parallel \text{AttitudeState(uncontrolled)} \setminus \{\text{attitude.setval}\})\)
\((\text{power.load.switch.adcs, attitude.getval, attitude.trans})\)]

\(\text{StateTelemetryStream(power.load.switch.adcs, attitude.getval, attitude.trans,}\)
\(\text{tlm.adcs_stream.setval, tlm.adcs_stream.getval,}\)
\(\text{tlm.adcs_stream.trans, fID})) \setminus \{\text{tlm.adcs_stream.setval}\})\)

The initial state of ComplexADCS\(_1\) is an unpowered mode, in which commands have no effect. Switching on power causes the ADCS to enter the detumble mode, and to produce a rate_nulled attitude state. In all of the powered ADCS modes, received mode commands are accepted if they appear in the the AllowedModeCmd set for the active mode, and otherwise rejected. Accepted mode commands produce a corresponding change in the attitude state.

In addition, science_active mode enables a function which provides direct commanding of the attitude. The process model also includes an attitude state, and a state telemetry stream which is active whenever the ADCS is powered. A diagram of the internal structure of the process model appears in fig. 5.17. □
Including Power

Different ADSCS modes may require the employment of different actuators or sensors. For example, an earth-pointing mode may necessitate the use an accurate earth-sensor, while other modes can achieve acceptable performance without the earth-sensor. These differences in actuator and sensor usage may translate into variations in subsystem power consumption across different modes. If these variations in power consumption have implications for the behavior of other subsystems, then the power interface of the subsystem model should include events to represent key changes in the level of power consumed.

The generic SubsysModePower process (definition 37) provides a way to add mode-based power behavior to an existing subsystem process model. The SubsysModePower process responds to mode transition events by generating power load transition events which reflect the difference in power consumption between the old mode and the new mode. These load transition events may in turn trigger changes in the behavior of other subsystems.

Example 5.4.2.4. Assume that the ADSCS represented by ComplexADCS \(_1\) consumes different quantities of power in different modes, and that these differences have some significance for other subsystems. In order to add a SubsysModePower process to the ComplexADCS \(_1\)

---

**Fig. 5.17:** The ADSCS model of example 5.4.2.3.
process model, it is first necessary to define a function which maps ADCS modes to their corresponding levels of power consumption. We represent power consumption in each mode using integer values, which we assume to be derived from a standard conceptual-level spacecraft power budget. The function $f_{\text{ModePower}}$ specifies the mapping.

$$f_{\text{ADCSModePower}}: \text{ADCSmode} \rightarrow \text{Integer}$$

$$f_{\text{ADCSModePower}} = \{(\text{unpowered}, 0),
(\text{detumble}, 10),
(\text{safehold}, 15),
(\text{science\_standby}, 15),
(\text{science\_active}, 20)\}$$

The $\text{ComplexADCS}_2$ process (fig. 5.18) combines $\text{ComplexADCS}_1$ with the generic $\text{SubsysModePower}$ process, synchronizing the two processes on ADCS mode transition events which are signaled on $\text{adcs\_modetrans}$.

$$\text{ComplexADCS}_2 = (\text{ComplexADCS}_1
\parallel\{\text{adcs\_modetrans}\})
\text{SubsysModePower}(\text{unpowered, adcs\_modetrans, power\_load\_delta\_adcs, } f_{\text{ADCSModePower}}))$$

The $\text{SubsysModePower}$ process is initialized in the $\text{unpowered}$ mode, and generates $\text{power\_load\_delta\_adcs}$ events based on the mapping defined by $f_{\text{ADCSModePower}}$. For example, an initial transition from the $\text{unpowered}$ mode to the $\text{detumble}$ mode triggers the event $\text{power\_load\_delta\_adcs.10}$, since the difference between the power consumed in $\text{unpowered}$ and the power consumed in $\text{detumble}$ is 10. A transition from $\text{detumble}$ back to $\text{unpowered}$ would produce the event $\text{power\_load\_delta\_adcs. - 10}$. ■
Incorporating Faults

Unfortunately, spacecraft subsystems do not work perfectly all of the time. It is therefore prudent to think about, and specify, the behavior of each subsystem in the face of internal faults. As described in section 5.3.3, we represent the occurrence of a fault as an abstract specification event. We incorporate faults into subsystem behavior models using the fault-tolerance modeling approach prescribed in sec. 12.3 of Roscoe’s text [3], in which faults events are treated as deterministic from the perspective of the process containing the fault, and the occurrence of fault events is regulated by an external process which encodes some hypothesis about the existence and quantity of fault events.

There are at least three different kinds of behavior a subsystem may exhibit in the presence of faults:

1. Termination of some or all functions on the first occurrence of a given fault (example 5.4.2.5).
2. Termination of some or all functions after more than one occurrence of a given fault; i.e. the subsystem is fault-tolerant (example 5.4.2.6).

3. Transition to a different mode of behavior in an attempt to contain or mitigate the effects of a fault (example 5.4.2.7).

We now consider examples of each of these three kinds of behavior.

Example 5.4.2.5. Suppose that the ADCS described by the SimpleADCS\textsubscript{1} process is a single-string design, and is consequently susceptible to complete subsystem failure in the event of an internal subsystem hardware failure. We further assume that hardware failures only occur when the subsystem is powered, and that the result of a subsystem failure is an uncontrolled attitude. A SimpleADCS\_with\_Failure process which models this behavior can be derived from the SimpleADCS\textsubscript{1} process by modifying the Controller(on) process (defined within SimpleADCS\textsubscript{1}, and describing the behavior of the ADCS when powered) to include the possibility of engaging in the fault event adcs\_fault\_hardware\_failure (defined in example 5.3.3.1). The new definition of Controller(on) is:

\[
\text{Controller(on)} = \\
\text{cmd.atcmd.set\_attitude?a } \rightarrow \text{attitude.setval!a } \rightarrow \text{Controller(on)} \\
\] 

\[
\text{power.load\_switch.adcs.off } \rightarrow \text{attitude.setval!uncontrolled } \rightarrow \text{Controller(off)} \\
\] 

\[
\text{adcs\_fault\_hardware\_failure } \rightarrow \text{attitude.setval!uncontrolled } \rightarrow \text{Failed} \\
\]

Upon occurrence of a fault, the new Controller(on) process enters the Failed state, which is defined by the process

\[
\text{Failed } = \text{cmd.atcmd.set\_attitude?a } \rightarrow \text{Failed} \\
\]

\[
\text{power.load\_switch.adcs?o } \rightarrow \text{Failed} \\
\]
Once in the Failed state, commands and power switching events have no effect, and no event can produce a transition out of Failed.

**Example 5.4.2.6.** Now suppose that the SimpleADCS ADCS is robust to a single hardware failure, but cannot tolerate more than that. Such behavior might be exhibited by a design which uses a traditional, redundant 4-reaction-wheel configuration to control the spacecraft attitude. We again assume that hardware failures only occur when the subsystem is powered, and that the result of a subsystem failure is an uncontrolled attitude.

Defining the SimpleADCS_with_FaultTolerance process requires modifying both the Controller(off) and Controller(on) processes within SimpleADCS, since to model fault tolerance we need to track the number of faults which have occurred. The Failed process retains the definition given it in example 5.4.2.5. The new definitions of Controller(off) and Controller(on) are:

\[
\text{Controller}(\text{off}, n\_faults) = \\
\text{cmd.attcmd.set_attitude?}a \rightarrow \text{Controller}(\text{off}, n\_faults) \\
\text{□} \\
\text{power.load.switch.adcs.on} \rightarrow \text{Controller}(\text{on}, n\_faults) \\
\text{Controller}(\text{on}, n\_faults) = \\
\text{cmd.attcmd.set_attitude?}a \rightarrow \text{attitude.setval!}a \rightarrow \text{Controller}(\text{on}, n\_faults) \\
\text{□} \\
\text{(power.load.switch.adcs.off} \rightarrow \text{attitude.setval!uncontrolled} \\
\rightarrow \text{Controller}(\text{off}, n\_faults)) \\
\text{□} \\
\text{(n\_faults} \geq 1) \& (\text{adcs\_fault.hardware\_failure} \\
\rightarrow \text{attitude.setval!uncontrolled} \rightarrow \text{Failed}) \\
\text{□} \\
\text{(n\_faults} < 1) \& \text{adcs\_fault.hardware\_failure} \rightarrow \text{Controller}(\text{on}, n\_faults + 1)
\]

where the initial state of the Controller process is assumed to be Controller(off, 0).
Following the approach to modeling and analyzing fault tolerance laid out in Roscoe’s text, we define a regulating process which encodes our “fault hypothesis,” i.e. the assumptions we wish to make about how many fault events might plausibly occur over the course of the spacecraft mission. The regulating process acts to limit the number of fault events that can occur, and is defined as

\[
\text{Fault}(n) = \begin{cases} 
\text{adcs\_fault\.hardware\_failure} \rightarrow \text{Fault}(n - 1) & \text{if } n > 0 \\
\text{Stop} & \text{else}
\end{cases}
\]

Roscoe’s technique for verifying fault tolerance involves composing the fault regulating process with the process to be verified, lazily abstracting (see [3]) the fault events, and refinement checking the resulting abstracted composite process against a version of the process to be verified which has fault events suppressed. Performing such a refinement check on the \textit{SimpleADCS\_with\_Failure} defined in example 5.4.2.5 using FDR demonstrates that

\[
\mathcal{Z}_{FD} \mathcal{L}_{\{\text{adcs\_fault}\}}(\text{SimpleADCS\_with\_Failure} [ \{\text{adcs\_fault}\} ] | \text{Fault}(1))
\]

That is, as expected, the behavior of \textit{SimpleADCS\_with\_Failure} when at least one fault is permitted to occur is not the same as that of \textit{SimpleADCS\_with\_Failure} when no faults can occur.

In contrast,

\[
\mathcal{E}_{FD} \mathcal{L}_{\{\text{adcs\_fault}\}}(\text{SimpleADCS\_with\_FaultTolerance} [ \{\text{adcs\_fault}\} ] | \text{Fault}(1))
\]

meaning that the behavior of \textit{SimpleADCS\_with\_FaultTolerance} does not change when a single fault is allowed to occur. The process is robust to single faults.
Of course,

\[
\begin{align*}
(SimpleADCS\_with\_FaultTolerance \parallel \{\text{adcs\_fault}\} \parallel \text{Stop}) \\
\mathbb{Z}_FDL_{\{\text{adcs\_fault}\}}(SimpleADCS\_with\_FaultTolerance \parallel \{\text{adcs\_fault}\} \parallel \text{Fault}(2))
\end{align*}
\]

since \textit{SimpleADCS\_with\_FaultTolerance} is robust to only a single fault. If, over the duration of the mission, two or more fault events can occur, the subsystem will fail. ■

**Example 5.4.2.7.** Figure 5.11 on page 106 is an example of an informal specification of a fault response involving a change in behavior. In the specification shown in that figure, the detection of a particular fault results in a transition to a different ADCS operating mode. We can produce a similar, but more formal, specification by modifying the \textit{Controller} process defined within the \textit{ComplexADCS$_1$} model.

In constructing the modified controller process, we assume that the ADCS fault handling mechanism reports the occurrence of any faults to other subsystems via the ADCS telemetry interface. Reporting faults in this way gives other subsystems a chance to change their own behavior to accommodate the altered ADCS behavior caused by the fault, or to make some attempt to remedy the fault. To incorporate fault reporting, we modify the definition of the \textit{ADCS\_tlm} datatype to accommodate a new kind of message, the \textit{adcs\_fault}.

\[
\text{datatype } ADCS\_tlm = \text{adcs\_ack\_cmd}.\text{CommandableADCSmode} \\
| \text{adcs\_reject\_cmd}.\text{CommandableADCSmode} \\
| \text{adcs\_stream}.\text{StateIF}.\text{AttitudeTlm} \\
| \text{adcs\_fault}.\text{ADCSFaultEvent}
\]

\[
\text{channel } tlm : ADCS\_tlm
\]

We assume that the fault for which we are specifying a response (violation of a Sun-Earth pointing constraint necessary for the well-being of a scientific instrument) can only occur in the \textit{science\_standby} and \textit{science\_active} modes. The auxiliary function \textit{SafingMode}
specifies, for a given mode, the mode to which the ADCS transitions if a fault occurs:

\[
\text{SafingMode} (\text{science\_active}) = \text{science\_standby}
\]

\[
\text{SafingMode} (\text{science\_standby}) = \text{safehold}
\]

The modified Controller process which incorporates the fault response specification is:

\[
\text{Controller} (m) =
\]

\[
(power\_load\_switch.adcs.on \rightarrow \text{Controller}(m))
\]

\[
(power\_load\_switch.adcs.off
\rightarrow \text{adcs\_modetrans.begin!unpowered} \rightarrow \text{attitude.setval!uncontrolled}
\rightarrow \text{adcs\_modetrans.end!unpowered} \rightarrow \text{Controller(\text{unpowered})})
\]

\[
(cmd\_modecmd?m' : \text{AllowedModeCmd}(m)
\rightarrow \text{tlm!adcs\_ack\_cmd.m'} \rightarrow \text{adcs\_modetrans.begin!m'}
\rightarrow \text{attitude.setval!f\text{ModeAtt}(m')} \rightarrow \text{adcs\_modetrans.end!m'}
\rightarrow \text{Controller(m')})
\]

\[
(cmd\_modecmd?m' : \text{DisallowedModeCmd}(m)
\rightarrow \text{tlm!adcs\_reject\_cmd.m'} \rightarrow \text{Controller(m')})
\]

\[
((m \in \{\text{science\_active, science\_standby}\}) \&
\text{adcs\_fault.sun\_earth\_violation} \rightarrow \text{tlm.adcs\_fault.sun\_earth\_violation}
\rightarrow \text{let m' = SafingMode(m)}
\rightarrow \text{within}\]

\[
\text{adcs\_modetrans.begin!m'} \rightarrow \text{attitude.setval!f\text{ModeAtt}(m')}
\rightarrow \text{adcs\_modetrans.end!m'} \rightarrow \text{Controller(m'))}
\]
This process states that the sun-earth-violation fault is assumed to occur only in the science-active and science-standby, that the detection of a fault is immediately reported as a telemetry message, and that the ADCS adopts a new mode in response to the fault. In this case, the detection of a fault (represented by the occurrence of a fault event) has been defined as part of an external choice among several other kinds of events. Thus, as written, the specification implies that the ADCS will not respond to faults while a command is in the midst of being acted upon. If this is not the intended behavior, then the specification would need to be rewritten, for example by using the interrupt operator to allow fault events to occur at any time during the response to a command event.

5.4.3 Command and Data Handling

The command and data-handling (CDH) subsystem is the focal point of spacecraft system behavior. It provides for spacecraft control and reconfiguration, and also collects and stores mission and housekeeping data. In most spacecraft designs, all of the other subsystems interface with the CDH subsystem, receiving commands from it, and passing data to it. We consider the problems of commanding and data storage separately, although the resulting models can, and should, be composed into a single CDH subsystem model.

Command and Control

Since CSP was originally developed for modeling and analyzing software systems, the command and control elements of the CDH subsystem are perhaps the most readily suited of any part of the spacecraft to modeling using CSP. Although this natural affinity makes the modeling task easier, it is still necessary to consider exactly how CSP should be applied to modeling typical spacecraft command and control systems.

Our approach in modeling the command and control portion of the CDH subsystem is to abstract from low-level operations which directly control individual hardware components, and to focus on high-level commands which may correspond to a sequence of several low-level hardware changes. For example, a high-level command to fire a thruster may actually involve a series of low-level operations to switch on various catalyst heaters, and open redundant thruster valves. However, explicating such a sequence of low-level operations is
unlikely to be possible during the early system design phases where we expect the modeling techniques described here to be of the most use, since the hardware design is unlikely to have reached that level of detail. Nor is it desirable from a modeling perspective, since extended sequences of low-level events are likely to result in an explosion of model states when the CDH model is composed with other subsystem models, making analysis of the system model intractable.

Spacecraft commanding is typically carried out through a mixture of real-time commands which are executed immediately upon receipt, and stored command sequences which may be triggered by a real-time command, timer, or some other event. Figure 5.19 is an example of a typical command handling requirement, excerpted from the WIRE System Requirements [24]. Similar requirements can be found in the ACE Spacecraft Design Specification [22] and the MGS Spacecraft Requirements [23], among others.

The core of our CDH models is a command processor which translates individual commands and events into corresponding actions, or commands to other subsystems. This command processor models the real-time, or immediate, commanding aspect of the CDH system. A simple command processor consists of an external choice over the commands and events to which the CDH is capable of responding. More complex command processors may provide different behavior in different subsystem modes. We model such mode-based behavior using the same approach as previously described for modeling mode-based behavior in an ADCS, such as that used in example 5.4.2.3.

**Example 5.4.3.1.** In this example we model an extremely simple command processor model for a scientific spacecraft. The command processor consists of a basic command decoder/router, which translates received spacecraft commands into corresponding subsystem commands. The processor is also capable of responding to ADCS fault events by placing the science payload into a standby mode, presumably in an effort to protect the instrument from adverse attitudes.

In this simple model the possible spacecraft commands consist only of commands to start and stop science operations, and commands to switch on and off certain subsystems.
CDH.REQ.GEN.7 COMMAND DISTRIBUTION

The C&DH system shall distribute a real-time command to the appropriate subsystem prior to the complete reception of the next command. Additionally, the C&DH system shall distribute stored commands at a maximum rate of ten per second.

In this example, we also include an “invalid” command alternative to model the reception of fragmentary or incorrectly formatted commands, which allows us to define the response of the CDH to the reception of these messages.

```plaintext
datatype SpacecraftCommand = start_science | stop_science | switch.Subsystem.OnOff | invalid

channel cmdin : SpacecraftCommand
```

We assume that all subsystem commands and all subsystem telemetry are communicated over a system bus, as described in section 5.3.1. We present only the subsystem commands and telemetry relevant to this example:

```plaintext
datatype PayloadCmd = pl_active | pl_standby

datatype DownlinkMsg = reject_cmd | ···

datatype EPSCmd = load_switch.Subsystem.OnOff | ···

datatype SubsysCmd = pl_cmd.PayloadCmd | dl_cmd.DownlinkMsg | eps_cmd.EPSCmd

datatype SubsysTlm = adcs_tlm.ADCSFaultEvent | ···

nametype SysBus = SubsysCmd ∪ SubsysTlm

channel systembus : SysBus
```
The actual command processor model, \textit{CommandProcessor}_1, is an extremely straightforward external choice over \textit{cmdin} events, along with the ADCS fault events that we assume the processor responds to:

\[
\text{CommandProcessor}_1 = \\
(\text{cmdin}.\text{invalid} \rightarrow \text{systembus}.\text{dl}_\text{cmd}!\text{rejectcmd} \rightarrow \text{CommandProcessor}_1) \\
\Box \\
(\text{cmdin}.\text{start}\_\text{science} \rightarrow \text{systembus}.\text{pl}_\text{cmd}.\text{pl}_\text{active} \rightarrow \text{CommandProcessor}_1) \\
\Box \\
(\text{cmdin}.\text{stop}\_\text{science} \rightarrow \text{systembus}.\text{pl}_\text{cmd}.\text{pl}_\text{standby} \rightarrow \text{CommandProcessor}_1) \\
\Box \\
(\text{cmdin}.\text{switch}\?x \rightarrow \text{systembus}.\text{eps}_\text{cmd}.\text{load}\_\text{switch}\!x \rightarrow \text{CommandProcessor}_1) \\
\Box \\
(\text{systembus}.\text{adcs}\_\text{tlm}.\text{adcs}\_\text{fault}\?f \\
\rightarrow \text{systembus}.\text{pl}_\text{cmd}.\text{pl}_\text{standby} \rightarrow \text{CommandProcessor}_1)
\]

In this case, the behavior specified for the command processor does not distinguish \textit{which} ADCS fault event has occurred, but responds the same way to \textit{any} ADCS fault.

Beyond simple command processors, more complex command and control systems add the capability to store sequences of commands (see fig. 5.20, excerpted from the \textit{WIRE CDH FSW Requirements Specification} [119]), and to execute predefined procedures which may include conditional logic [120]. Although stored command sequences are, for the most part, relatively easy to model in CSP as sequences of events, there are two issues which require some additional consideration. Namely, time-tagging of individual commands, and storage of arbitrary command sequences.

In many spacecraft designs, the commands within a stored sequence are tagged with either an absolute or relative time at which the command should be executed [22–24, 120]. However, in developing CSP models of spacecraft behavior we are largely concerned with temporal ordering, rather than the details of timing, which makes including direct modeling
The flight software shall autonomously process individual commands from the relative time-tagged command buffers.

The flight software shall permit the relative time command sequences to execute concurrently.

The flight software shall initiate a relative time command sequence based on:

- receipt of a RTS control command from the ground,
- request from another flight software subsystem,
- an absolute time-tagged command, and
- a relative time-tagged command.

Fig. 5.20: Specification of stored-command handling for WIRE.

of time-tagged commands problematic (although not impossible). Fortunately, time-tags in stored sequences are largely used as a mechanism for establishing the execution sequence of commands [121], rather than as a way to define timing critical operations. As a result, our modeling of stored sequences abstracts from time-tags, and simply specifies the order of the commands in a sequence. This has the additional advantage that we can model and understand the impact of particular command sequences even when the design has not yet reached a stage where the precise timing of the commands can be specified. Where timing is particularly important for the analysis of a command sequence, or the interaction between several command sequences, we can add temporal constraints such as those described in section 4.3.2. However, critical timing requirements are probably better analyzed using tools specifically intended for the analysis of timed systems, such as UPPAAL [122].

Example 5.4.3.2. A command sequence intended to first switch a payload into standby mode, and then to power it down, might be represented by the abstracted event sequence

\[
\langle \text{systembus.pl_cmd.pl_standby, systembus.eps_cmd.load_switch.pl.off} \rangle
\]
Spacecraft designs that include stored command sequences usually provide the capability for ground controllers to upload new stored sequences to the spacecraft for later execution, giving the spacecraft a certain amount of operational flexibility. While this is convenient for spacecraft operators, it is less so for those of us wishing to analyze spacecraft behavior. To begin with, it is very difficult to predict all of the possible command sequences that a spacecraft operator might choose to upload to the spacecraft, so we must content ourselves with modeling only those sequences which seem most likely to be used in the mission under design. Of more immediate concern, attempting to directly model the upload and buffering of stored sequences is infeasible: buffer processes with capacities sufficient to hold sequences of more than a few commands result in intractably large state-spaces.

Instead of trying to model command uploading and storage directly, we adopt the approach of modeling as sequential processes specific sequence uploads that are likely to be used during the mission under design, and including those processes within the CDH subsystem model. The command sequence processes are composed in parallel with a controller, which moderates execution of the stored-sequence processes (fig. 5.21).

Fig. 5.21: Controlling the execution of stored sequence processes.
We model uploading of a given sequence as communication of a token which represents the “uploaded” sequence from the ground station to the CDH subsystem. The set of tokens held at any given time by the execution controller represents the command sequences presently stored in the CDH. The execution controller refuses to execute stored sequences for which it does not presently hold a token. This approach to modeling sequence uploads is similar to the way in which mobile channels were previously modeled by the author [123].

Figure 5.22 illustrates the token-based approach to modeling sequence-loading. Initially the execution controller holds the token 1, and only the process representing stored sequence 1 is considered “loaded.” Commands to run sequence 1 will result in the process representing that sequence being triggered, while commands to run any of the other sequences will be rejected since those sequences are not considered to have yet been loaded into the sequence store. Once the execution controller receives the token 2, it begins to treat stored sequence 2 as having been loaded, allowing the process representing that sequence to also be triggered. Example 5.4.3.3 illustrates the use of token-based modeling of stored sequences.

Fig. 5.22: Modeling sequence-loading using tokens.
Example 5.4.3.3. For the purposes of this example, we assume that there are three possible stored sequences that might be loaded onto the spacecraft being modeled, and define a datatype to refer to these sequences accordingly:

\[
\text{datatype } \text{SequenceName} = \text{seq.}\{1 \ldots 3\}
\]

We use the values in the datatype \text{SequenceName} as the tokens which represent loading and unloading of sequences.

Command sequences may be loaded, unloaded, run, or stopped. Commands to perform each of these actions are received by the execution controller over the channel \text{cmdin}. The sequence to which a given command applies is determined by the token associated with that command.

\[
\text{datatype } \text{SeqCmd} = \text{load}_\text{seq} | \text{unload}_\text{seq} | \text{run}_\text{seq} | \text{stop}_\text{seq}
\]

\[
\text{channel } \text{cmdin} = \text{SeqCmd}.\text{SequenceName}
\]

Internally, the execution controller may either run a loaded sequence, or terminate an already running sequence:

\[
\text{datatype } \text{SeqOp} = \text{run} | \text{terminate}
\]

\[
\text{channel } \text{seq}_\text{exec} : \text{SequenceName}.\text{SeqOp}
\]

We define three processes to represent the three stored sequences which may be loaded. The first sequence, \text{CommandSeq}_1, commands the ADCS to the \text{science-active} mode, and switches the payload to its \text{active} mode. Execution of the sequence is triggered by reception of a \text{run} command from the execution controller, and may be interrupted at any time by reception of a \text{terminate} command.
\textit{CommandSeq}_1 = ((\textit{seq\_exec}\textunderscore\textit{seq}\textunderscore1\textunderscore\textit{run} \\
\quad \rightarrow \textit{EventSeq}((\textit{systembus.adcs\_cmd.modecmd.science\_active}, \\
\quad \textit{systembus.pl\_cmd.pl\_active})) \\
\qquad \triangle \textit{seq\_exec}\textunderscore\textit{seq}\textunderscore1\textunderscore\textit{terminate} \rightarrow \text{Skip}); \textit{CommandSeq}_1

The second and third sequences follow a similar format to \textit{CommandSeq}_1. \textit{CommandSeq}_2 switches the payload into \textit{standby} mode, and then powers the payload down. \textit{CommandSeq}_3 is a little more complex than the other two sequences, in that it includes conditional logic. \textit{CommandSeq}_3 checks the current attitude state as reported by the ADCS, and, if the attitude is one of scientific relevance, switches the payload into \textit{active} mode.

\textit{CommandSeq}_2 = 

\((\textit{seq\_exec}\textunderscore\textit{seq}\textunderscore2\textunderscore\textit{run} \\
\quad \rightarrow \textit{EventSeq}((\textit{systembus.pl\_cmd.pl\_standby}, \\
\quad \textit{systembus.eps\_cmd.load\_switch.pl\_off})) \\
\qquad \triangle \textit{seq\_exec}\textunderscore\textit{seq}\textunderscore2\textunderscore\textit{terminate} \rightarrow \text{Skip}); \textit{CommandSeq}_2

\textit{CommandSeq}_3 = 

\((\textit{seq\_exec}\textunderscore\textit{seq}\textunderscore3\textunderscore\textit{run} \rightarrow \textit{systembus.adcs\_tlm.adcs\_stream.getval}\?a \\\n\quad \rightarrow \text{if } a \in \{\text{science\_target}\} \\\n\quad \quad \text{then } \textit{systembus.pl\_cmd.pl\_active} \rightarrow \text{Skip} \text{ else Skip}) \\
\qquad \triangle \textit{seq\_exec}\textunderscore\textit{seq}\textunderscore3\textunderscore\textit{terminate} \rightarrow \text{Skip}); \textit{CommandSeq}_3

The three stored sequence processes are composed in parallel with the execution controller, which manages loading, unloading, and running of the stored sequences:

\textit{StoredSequenceProcessor} = 

\(((\textit{ExecutionControl}\|\|\textit{seq\_exec}\textunderscore\textit{seq}\textunderscore1) \| \textit{CommandSeq}_1) \\
\quad \|\|\|\textit{seq\_exec}\textunderscore\textit{seq}\textunderscore2 \| \textit{CommandSeq}_2 \\
\quad \|\|\|\textit{seq\_exec}\textunderscore\textit{seq}\textunderscore3 \| \textit{CommandSeq}_3) \setminus \{\textit{seq\_exec}\} \)
The execution controller itself is defined as an external choice over sequence commands. The tokens representing currently loaded sequences are kept in the set $\text{Loaded}$, which is initially empty (i.e. no sequences are loaded). A $\text{load_seq}$ command causes a new token to be added to the $\text{Loaded}$ set, while an $\text{unload_seq}$ command removes the designated token from the $\text{Loaded}$ set. Decisions about whether or not to allow a sequence to be executed are made based on the current membership of the $\text{Loaded}$ set.

$$\text{ExecutionControl} =$$
let

$$EC(\text{Loaded}) =$$

$\text{cmdin.load_seq}s \rightarrow EC(\text{Loaded} \cup \{s\})$

$\text{cmdin.run_seq}s : \text{Loaded} \rightarrow \text{seq_exec.s.run} \rightarrow EC(\text{Loaded})$

$\text{cmdin.run_seq}s : (\text{SequenceName} \setminus \text{Loaded})$

$\rightarrow \text{systembus.dl_cmd.rejectcmd} \rightarrow EC(\text{Loaded})$

$\text{cmdin.unload_seq}s : \text{Loaded}$

$\rightarrow \text{seq_exec.s.terminate} \rightarrow EC(\text{Loaded} \setminus \{s\})$

$\text{cmdin.unload_seq}s : (\text{SequenceName} \setminus \text{Loaded})$

$\rightarrow \text{systembus.dl_cmd.rejectcmd} \rightarrow EC(\text{Loaded})$

$\text{cmdin.stop_seq}s \rightarrow \text{seq_exec.s.terminate} \rightarrow EC(\text{Loaded})$

within

$$EC(\emptyset)$$

As an example of how the composite $\text{StoredSequenceProcessor}$ operates, consider the following trace, which first attempts to execute sequence 1 before it has been loaded, then
loads the sequence and executes it. We use $\tau (ev)$ to represent a hidden occurrence of the event $ev$.

```plaintext
cmdin.run_seq.seq.1
systembus.dl_cmd.rejectcmd
cmdin.load_seq.seq.1
cmdin.run_seq.seq.1
$\tau (seq_exec.seq.1.run)$

systembus.adcs_cmd.modecmd.science_active
systembus.pl_cmd.pl_active
```

Stored commands provide a limited amount of onboard autonomy, allowing a spacecraft to undertake complex operations when out of contact with a ground station. Beyond simple sequences, several kinds of autonomous spacecraft control are also possible, including rule-based reactions to events, and onboard planning to manage resource allocation between different tasks. However, detailed modeling of those kinds of autonomy is both beyond the scope of the present work, and to a certain extent already addressed by existing research. For example, research by O’Halloran and McEwan on expressing rule-based control systems in CSP [124] is a good candidate for modeling rule-based autonomy. Similarly, work by Smith et al. on the use of Promela and SPIN to model and verify autonomous planners [32] is, given the similarities between Promela and CSP, likely to be easy to translate into CSP.

**Data Handling**

Data handling is largely concerned with the aggregation and storage of housekeeping and payload telemetry. But, just as direct modeling of stored commands sequences in terms of classical CSP buffer processes results in intractably large state-spaces, so too does direct modeling of data storage. Moreover, there are tools other than CSP which are far better suited to computing onboard data buffer usage profiles.
Instead of attempting to directly model data storage, we consider an abstract view of the data-handler which focuses on those aspects of data-handling that are most relevant to defining subsystem and system behavior. Figures 5.23 (excerpted from the ACE Spacecraft Design Specification [22]) and 5.24 (excerpted from the WIRE CDH FSW Requirements Specification [119]) exemplify the type of information that we are interested in capturing within the process model of a data-handler: the behavior of the handler in different operating modes, how the handler transitions between those modes, and how the handler reacts to reaching a resource limit. These aspects of data-handling behavior are independent of the actual data values being stored, which allows us to abstract from individual items of data.

6.2.11. Data Recorders

Recorder operational modes shall be:

- record
- reproduce
- set record pointer
- set reproduce pointer
- standby

Fig. 5.23: Specification of data storage modes for ACE.

401.5 The flight software shall provide via ground command two modes of bulk memory data storage when a partition is full,

- discard any new data for the full partition, and
- continue storing new data, overwriting the oldest data in that partition.

401.6 A system event shall be generated when the data storage area for a particular partition is determined to be full.

Fig. 5.24: Specification of data-handling for the WIRE spacecraft.
At an intermediate level of abstraction, we can define data-handling models which ignore the content of received data messages, but explicitly track the number of messages, and thus the quantity of data that is stored at any given time. Models of this sort are most useful when the production of the data to be stored is modeled as discrete messages, and knowledge of the quantity of data that has been stored is important. The latter condition may be important in systems where avoidance of a storage overflow relies on assumptions about how other subsystems operate, rather than on a controller which senses the state of the storage resource and switches data sources on and off accordingly.

Models of the sort just described are essentially counters, which increment or decrement their value whenever a message is received or transmitted. However, in addition to the basic counting behavior, the model of the data-handler must also define the data-handler modes of behavior, the mode transitions, and the responses (if any) to reaching counter values which represent key quantities of stored data.

**Example 5.4.3.4.** A data-handling system receives discrete packets of payload data to be stored, and can store up to \(N\) packets. The data-handler can be commanded into two modes: a standby mode, in which it neither stores nor provides data, and an active mode, in which it is capable of performing both packet read and packet write operations (contrast this behavior with the example in fig. 5.23, in which there are separate modes for reading and writing). The data-handler generates a system event when its store becomes full.

The types and channels which represent the explicit interfaces implied by the preceding description of the data-handler are:

```plaintext
datatype SystemEvent = data_store_full | ···
datatype DownlinkMsg = no_data | pl_data | system_event.SystemEvent | ···
datatype SubsysCmd = dl_cmd.DownlinkMsg | ···
datatype SubsysTlm = pl_tlm.Measurement | ···
nametype SysBus = SubsysCmd \cup SubsysTlm
channel systembus : SysBus
```
In addition to the explicit interfaces, the model incorporates an abstract specification event which signals a loss of data when packets are received after the data store has become full, and a specification-event channel which signals data-handler mode transitions.

The DiscreteDataHandler process models the behavior described above. The data-handler is assumed to be initially in standby mode, with no data stored. In active mode, the response to read_packet commands varies depending on whether or not there are any packets in the data store. Similarly, the response to receiving a new data packet depends on whether or not the data store is at capacity.

\[
\text{DiscreteDataHandler}(N) = \\
\text{let} \\
\text{Standby}(n) = \\
\quad (\text{cmd.modecmd.active} \rightarrow \text{dhd_modetrans.begin.active} \\
\quad \quad \rightarrow \text{dhd_modetrans.end.active} \rightarrow \text{Active}(n)) \\
\]

\[
\quad (\text{cmd.read_packet} \rightarrow \text{Standby}(n)) \\
\]

\[
\quad (\text{systembus.pl_tlm.meas?m} \rightarrow \text{Standby}(n)) \\
\]

\[
\text{Active}(n) = \\
\quad (\text{cmd.modecmd.standby} \rightarrow \text{dhd_modetrans.begin.standby} \\
\quad \quad \rightarrow \text{dhd_modetrans.end.standby} \rightarrow \text{Standby}(n))
\]
(n > 0) & (cmd.read_packet
→ systembus.dr_cmd!pl_data → Active(n − 1))

(n ≤ 0) & (cmd.read_packet
→ systembus.dr_cmd!no_data → Active(n))

(n < N − 1) & (systembus.pl_tlm.meas?m → Active(n + 1))

(n = N − 1) & (systembus.pl_tlm.meas?m
→ systembus.dr_cmd!system_event.data_store_full → Active(n + 1))

(n ≥ N) & (systembus.pl_tlm.meas?m → packet_dropped → Active(n))

within

Standby(0)

Although we haven’t shown it here, a generic SubsysModePower could be added to
the model in this example, providing a way to capture the impact of differences in power
consumption between the active and standby modes.

Although the approach to modeling data-handlers just described will work in some
situations, it does suffer from two problems. First, the discrete, message-based way of
dealing with incoming data meshes poorly with our approach for modeling streams of data,
which is exemplified by the StateTelemetryStream behavior. Second, as the value assigned
to N is made larger, there is a corresponding growth in the state-space of the data-handler
model. At some point, the resulting state-space will become large enough to preclude timely
behavior analysis. One solution to both of these problems is to adopt an even more abstract
modeling approach.

We achieve a greater level of abstraction by making two changes to the way in which
the data-handler is modeled. First, instead of explicitly tracking the quantity of stored data,
we use three qualitative values to represent the state of the data storage resource: \textit{empty}, \textit{partially full}, and \textit{full}; additional qualitative states may be added if the data-handling behavior is expected to vary for different levels of partial storage resource consumption.

Second, instead of dealing with individual data messages, our stream-based data-handler models respond only to transitions between the \textit{inactive} stream state, and any active state. That is, whether or not a transition from one qualitative storage state can occur is dependent on the state of the stream providing data to be stored. For example, if the stream is inactive, then it doesn’t make sense for the storage state to be able to transition to a qualitative state representing greater consumption of storage resources.

A stream-based data-handler model specifies the data-handling behavior as a function of the three different qualitative states. As with the message-based data-handling models, the stream-based data-handler also defines the different modes of data-handling behavior, and the events which cause transitions between those modes.

**Example 5.4.3.5.** A data-handling system stores a stream of payload data. The data-handler can be commanded into three modes: a standby mode, in which it neither stores nor provides data, and a read mode, in which it responds to request for stored data by sending a data packet to the downlink, and a write mode, in which it stores streaming data.

We first define datatypes to represent the qualitative storage states, and to help keep track of whether or not the data stream is active or inactive.

```plaintext
datatype StorageState = store_empty | store_partial | store_full

datatype StreamState = stream_inactive | stream_active
```

The types and channels which represent the explicit interfaces for the data-handler are:

```plaintext
datatype DownlinkMsg = no_data | pl_data | ···

datatype SubsysCmd = dl_cmd.DownlinkMsg | ···

datatype SubsysTlm = pl_tlm_stream.StateIF.MeasurementStream | ···

nametype SysBus = SubsysCmd \cup SubsysTlm
```
channel systembus : SysBus

datatype DataHandlerMode = standby | read | write

datatype DataHandlerCommand = read_packet | modecmd.DataHandlerMode

channel cmd : DataHandlerCommand

We also include specification events to signal loss of data, and data-handler mode transitions. As a convenience, we define a subset of data stream values which correspond to an active stream.

channel dhd_modetrans : ModeTrans.DataHandlerMode

channel data_loss

nametype ActiveStream = MeasurementStream \ {inactive}

The actual data-handler behavior is captured in the StreamDataHandler process. The data-handler is assumed to start in the standby mode, with an empty data store, and an inactive incoming data stream. In both the standby mode and the read mode a transition on the part of the incoming data stream to an active state results in data loss, since the incoming data is not stored in either of those modes.

In the read mode, the response to a request for a packet of data depends on the qualitative state of the data store. The new store state also depends on the current state. In particular, a read operation while the store is partially full can result either in a continuation of the partially full state, or a transition to the empty state. Since the actual size of the data store is undefined, the number of read operations required to produce a transition to the empty state is also undefined. We use a nondeterministic choice to represent this inherent uncertainty in the model. A similar line of reasoning leads to the nondeterministic choice which appears in the write mode.
StreamDataHandler =

let

Standby(store, stream) =

(cmd.modecmd.read → dhd_modetrans.begin.read
 → dhd_modetrans.end.read → Read(store, stream))

☐

(cmd.modecmd.write → dhd_modetrans.begin.write
 → dhd_modetrans.end.write → Write(store, stream))

☐

(systembus.pl_tlm_stream.trans?m : ActiveStream
 → data_loss → Standby(store, stream_active))

☐

(systembus.pl_tlm_stream.trans.inactive
 → Standby(store, stream_inactive))

Read(store, stream) =

(cmd.modecmd.standby → dhd_modetrans.begin.standby
 → dhd_modetrans.end.standby → Standby(store, stream))

☐

(cmd.modecmd.write → dhd_modetrans.begin.write
 → dhd_modetrans.end.write → Write(store, stream))

☐

(store = store_empty) & (cmd.read_packet
 → systembus.dl_cmd!no_data → Read(store, stream))

☐

(store = store_partial) & (cmd.read_packet
 → systembus.dl_cmd!pl_data
 → (Read(store_partial, stream) □ Read(store_empty, stream)))
\[
\text{Write}(\text{store}, \text{stream}_{-}\text{inactive}) =
\]
\[
(\text{cmd}.\text{modecmd}.\text{standby} \rightarrow \text{dhd}_-\text{modetrans}.\text{begin}.\text{standby} \rightarrow \text{dhd}_-\text{modetrans}.\text{end}.\text{standby} \rightarrow \text{Standby}(\text{store}, \text{stream}_{-}\text{inactive}))
\]
\[
(\text{cmd}.\text{modecmd}.\text{read} \rightarrow \text{dhd}_-\text{modetrans}.\text{begin}.\text{read} \rightarrow \text{dhd}_-\text{modetrans}.\text{end}.\text{read} \rightarrow \text{Read}(\text{store}, \text{stream}_{-}\text{inactive}))
\]
\[
\text{Write}(\text{partial}, \text{stream}_{-}\text{active})
\]
\[
\text{Write}(\text{full}, \text{stream}_{-}\text{active})
\]
\[
\text{Write}(\text{empty}, \text{stream}_{-}\text{active}) = \text{Write}(\text{partial}, \text{stream}_{-}\text{active})
\]
\[
\text{Write}(\text{full}, \text{stream}_{-}\text{active})
\]
\[ \text{Write}(\text{store}, \text{stream\_active}) = \\
(\text{cmd\_modecmd\_standby} \rightarrow \text{dhd\_modetrans\_begin\_standby} \\
\rightarrow \text{dhd\_modetrans\_end\_standby} \rightarrow \text{Standby}(\text{store}, \text{stream\_inactive})) \\
\Box \\
(\text{cmd\_modecmd\_read} \rightarrow \text{dhd\_modetrans\_begin\_read} \\
\rightarrow \text{dhd\_modetrans\_end\_read} \rightarrow \text{Read}(\text{store}, \text{stream\_inactive})) \\
\Box \\
(\text{systembus\_pl\_tlm\_stream\_trans\_inactive} \rightarrow \text{Write}(\text{store}, \text{stream\_inactive})) \\
\Box \\
(s = \text{store\_partial}) \& (\text{Write}(\text{store\_partial}, \text{stream\_active}) \\
\Box \\
(data\_loss \rightarrow \text{Write}(\text{store\_full}, \text{stream\_active}))) \\
\text{within} \\
\text{Standby}(\text{store\_empty}, \text{stream\_inactive}) \]

### 5.4.4 Electrical Power

The spacecraft electrical power subsystem (EPS), as its name suggests, is the source of electrical power for a spacecraft. Power may be generated in a wide variety of ways, although the most common configuration consists of a combination of solar arrays and rechargeable batteries, the latter providing power when the solar arrays are unable to do so. In addition to managing power generation, many spacecraft designs allocate responsibility for switching on and off the various spacecraft loads to the EPS. Some spacecraft designs also give the EPS responsibility for managing pyrotechnic events, such as mechanism deployments.

#### Power Source Models

In modeling how the EPS interacts with other subsystems, we are not particularly concerned with the specifics of how the EPS generates power. Rather, what matters is
the quantity of power available, and how the quantity varies in response to interactions with other subsystems. This allows us to treat the EPS as a black-box, and to abstract the power generation aspect of EPS behavior into a relatively simple model which captures system-level assumptions (or EPS design requirements) regarding the EPS load capacity.

Perhaps the simplest model of a power source consists of a source with a fixed load capacity. This corresponds to a fairly strict EPS design requirement which permits no variation in the quantity of power available to the electrical loads; the actual EPS must always be capable of supplying at least the amount of power specified by the fixed load capacity. A simple fixed-capacity model is easily defined using a quantitative resource process (see section 4.3.1).

**Example 5.4.4.1.** Consider a simple EPS behavioral specification, consisting of a fixed-capacity power source with a load capacity of \( \text{CAPACITY} = 20 \) integer-scaled units of power, and an associated telemetry stream. The process model is a composition of a \text{QuantResource} process, which represents the amount of power currently allocated to the various loads, and a \text{StateTelemetryStream} (fig. 5.25). The explicit interface for the EPS is defined as

```plaintext
nametype PowerRange = \{-CAPACITY .. CAPACITY\}
channel power : Power
datatype EPS_tlm : load_level.PowerRange
channel eps_tlm_stream : StateIF.EPS_tlm
datatype EPS_cmd : eps_tlm.OnOff
channel cmd : EPS_cmd
```

where \( \text{power} \) is the interface through which other subsystems communicate power consumption changes, \( \text{eps\_tlm\_stream} \) is the channel for the telemetry stream, and \( \text{cmd} \) is a channel through which commands to activate and deactivate the telemetry stream can be sent.

Internally, we require a channel for communications between the \text{StateTelemetryStream} process and the \text{QuantResource} process. We also require a channel for communicating load...
FixedCapacityEPS

\text{Quantitative Resource}

\text{State Telemetry Stream}

cmd.eps\_tlm

eps\_tlm\_stream

Fig. 5.25: The EPS model of example 5.4.4.1.

level changes to the \textit{QuantResource} process, since it does not understand the different subsystem names that appear in \texttt{power.load\_delta} events. We use renaming to bring this internal channel into correspondence with the EPS explicit interface defined above.

\begin{verbatim}
channel power_alloc : StateIF.PowerRange
channel power_delta : PowerRange

The \textit{FixedCapacityEPS} process combines the \textit{QuantResource} and \textit{StateTelemetryStream} processes. The \textit{QuantResource} process is permitted to take on a range of values between 0 and \texttt{CAPACITY}, and is set to an initial value of 0. The function \( f_{\text{EPStlm}}(l) = \text{load\_level.l} \) converts load quantities into telemetry values.

\textit{FixedCapacityEPS} =

\((\text{QuantResource(power\_delta, power\_alloc.getval, power\_alloc.trans,}

\quad \text{0, CAPACITY, 0)}

\quad | \{\{\text{power\_alloc}\}\})

\text{StateTelemetryStream(cmd.eps\_tlm, power\_alloc.getval, power\_alloc.trans,}

\quad \text{eps\_tlm\_stream.setval, eps\_tlm\_stream.getval,}

\quad \text{eps\_tlm\_stream.trans, f_{\text{EPStlm}}})

\quad \} \{\text{power\_alloc, eps\_tlm\_stream.setval}\})

\quad \} [s : \text{Subsystem} \bullet \text{power\_delta} \leftarrow \text{power\_load\_delta.s}]\end{verbatim}
When the *FixedCapacityEPS* process is composed with other subsystem process models, *power.load_delta* events generated by the subsystems will cause the value of the total allocated power to change. These variations will also be apparent in the telemetry stream produced by the EPS. For example, the sequence of *power.load_delta* events

\[
\text{power.load\_delta.adcs.5} \\
\text{power.load\_delta.cdh.3} \\
\text{power.load\_delta.adcs.\ -\ 2} \\
\text{power.load\_delta.payload.10}
\]

will result in the sequence of telemetry transition events

\[
\text{eps\_tlm\_stream.trans.load\_level.5} \\
\text{eps\_tlm\_stream.trans.load\_level.8} \\
\text{eps\_tlm\_stream.trans.load\_level.6} \\
\text{eps\_tlm\_stream.trans.load\_level.16}
\]

If the amount of allocated power exceeds *CAPACITY*, the *QuantResource* will produce a *resource\_overflow* exception-event.

The fixed-capacity EPS is conceptually very simple. However, not all spacecraft are designed with a fixed maximum load level, and may operate in different modes depending on the state of the power source upon which they rely. For example, some spacecraft designs cannot provide as much power during eclipse as they do during sunlit periods due to battery capacity limitations, and will operate in a reduced power mode (perhaps disabling certain payloads) during eclipse periods. Other designs, especially those with fixed (rather than sun-tracking) solar arrays, generate different quantities of power in different attitudes, and again may use different operating modes depending on the spacecraft attitude.

Modeling a variable-capacity power source is, not surprisingly, somewhat more complex than modeling a fixed-capacity source. In addition to tracking the allocated power via a
QuantResource process, it is necessary to track the available power, which will change as a result of implicit interface events such as attitude state changes. Whenever the levels of available or allocated power change, we must check to ensure that the allocated power (i.e. the total power being consumed by all subsystems) does not exceed the available power. A check of this sort can be accomplished using a process such as the one in the following example.

**Example 5.4.4.2.** DynamicCapacityCheck is a parameterized process suitable for checking the relationship between available and allocated power in an EPS process model. The process parameters are StateIF channels that provide an interface to the present allocated and available power states. Each transition in one of the states triggers a comparison of both state values.

\[
\text{DynamicCapacityCheck}(\text{power\_alloc}, \text{power\_avail}) = \\
\quad \text{let} \\
\quad \quad \text{Check}(pA, pL) = \text{if } pA < pL \\
\quad \quad \quad \text{then } \text{eps\_exception\_resource\_overflow } \rightarrow \text{STOP} \\
\quad \quad \quad \text{else } \text{DynamicCapacityCheck}(\text{power\_alloc}, \text{power\_avail}) \\
\quad \quad \text{within} \\
\quad \quad \quad (\text{power\_alloc\_trans}\?pL \rightarrow (\text{power\_avail\_getval}\?pA \rightarrow \text{Check}(pA, pL))) \\
\quad \quad \text{□} \\
\quad \quad \quad \text{power\_avail\_trans}\?pA \rightarrow \text{Check}(pA, pL)) \\
\quad \text{□} \\
\quad \quad \text{□} \\
\quad \quad \quad (\text{power\_avail\_trans}\?pA \rightarrow (\text{power\_alloc\_getval}\?pL \rightarrow \text{Check}(pA, pL))) \\
\quad \quad \text{□} \\
\quad \quad \quad \text{power\_alloc\_trans}\?pL \rightarrow \text{Check}(pA, pL)) \\
\]

In this version of the DynamicCapacityCheck process, if the amount of allocated power exceeds the amount of available power, the spacecraft is assumed to have entered an unsupportable state, and an exception-event is issued. Other variants of DynamicCapacityCheck
might instead issue commands to switch off certain non-essential loads when the allocated power is too high.

The external choice between getval and trans events in each branch of the process prevents a deadlock from occurring if both the allocated and available power states undergo a transition at the same time.

**Example 5.4.4.3.** As an example of a variable-capacity EPS, consider a spacecraft power subsystem which provides different amounts of power in different spacecraft attitudes. In this case, we assume that the (integer-scaled) quantities of power available in the different spacecraft attitudes are defined by the function

\[
\begin{align*}
\text{available}(\text{uncontrolled}) &= 8 \\
\text{available}(\text{sun_pointing}) &= 10 \\
\text{available}(\text{earth_limb_scan}) &= 10
\end{align*}
\]

where uncontrolled, sun_pointing, and earth_limb_scan represent the three different attitude states in which the spacecraft may find itself.

As with the FixedCapacityEPS, in the VariableCapacityEPS process (fig. 5.26) we model the amount of allocated power using a QuantResource process. In addition, we add an AvailablePower process to map spacecraft attitude states to quantities of available power, and make use of the DynamicCapacityCheck process defined in the previous example to carry out comparisons between the levels of allocated and available power.

![VariableCapacityEPS Diagram](image)

**Fig. 5.26:** The EPS model of example 5.4.4.3.
VariableCapacityEPS =

let

AvailablePower(a) =
(attitude.trans? a' → power_avail.trans! available(a') → AvailablePower(a'))

power_avail.getval! available(a) → AvailablePower(a)

AllocatedPower =
QuantResource(power_delta, power_alloc.getval, power_alloc.trans, 0, CAPACITY, 0)

INIT_ATTITUDE = uncontrolled

within

(((AllocatedPower || {power_alloc}]]
DynamicCapacityCheck(power_alloc, power_avail))
||{power_avail}]]
AvailablePower(INIT_ATTITUDE))
\ {power_alloc, power_avail}]

[s : Subsystem • power_delta ← power.load_delta.s]

The VariableCapacityEPS process behaves as follows:

- power.load_delta events cause the quantity of allocated power to change.
- attitude.trans events cause the quantity of available power to change.
- If the allocated power exceeds the available power, an exception-event is generated.

A similar approach to that used in the preceding example can be applied to model the different levels of available power in and out of eclipse, or indeed any other externally
induced variation in the level of available power. However, the more external influences that are added, the greater the state-space of the EPS process model will become. So in general it is preferable to limit the variability of the available power level within the process model as much as possible.

**Power-Switching Behavior**

In addition to providing a source of power, the spacecraft EPS is typically also responsible for managing the switching of power to different spacecraft loads. Power-switching is usually performed in response to commands from the spacecraft CDH subsystem, although other events may sometimes trigger autonomous action on the part of the EPS. Figure 5.27 contains an example description of the power switching behavior of an EPS, excerpted from the *WIRE System Requirements* [24].

A simple EPS power-switching behavior may consist of little more than translation of commands received from the CDH into load-switching events which represent the opening and closing of power relays. Such behavior is fairly straightforward to represent in CSP, either as a lifted function, or more directly as a process expression.

**Example 5.4.4.4.** Recall the fixed-capacity EPS of example 5.4.4.1. We now extend that EPS by adding a simple switching behavior. We first modify the definition of the $\text{EPScmd}$ type to include switch commands.

\[
\text{datatype } \text{EPScmd} = \text{eps}_\text{tlm}.\text{OnOff} | \text{sw}_\text{Subsystem}.\text{OnOff}
\]

A simple EPS power-switching behavior may consist of little more than translation of commands received from the CDH into load-switching events which represent the opening and closing of power relays. Such behavior is fairly straightforward to represent in CSP, either as a lifted function, or more directly as a process expression.

**Example 5.4.4.4.** Recall the fixed-capacity EPS of example 5.4.4.1. We now extend that EPS by adding a simple switching behavior. We first modify the definition of the $\text{EPScmd}$ type to include switch commands.

\[
\text{datatype } \text{EPScmd} = \text{eps}_\text{tlm}.\text{OnOff} | \text{sw}_\text{Subsystem}.\text{OnOff}
\]

...Commands for controlling the power distribution relays are received from the SCS [spacecraft computer system] over the 1553 data bus. The SPE [spacecraft power electronics] decodes the command and provides pulse shaping and current drive to the appropriate relays... Commands for controlling the power, solar array, and pyro relays are initiated by the SCS. The SPE command timer provides a one-time power “on” command to the ACE [attitude control electronics] power relays.

Fig. 5.27: EPS switch-command handling for the WIRE spacecraft.
The switching behavior is defined by the process *CommandLogic*. The *CommandLogic* process runs in parallel with the rest of the EPS model. We assume that the execution of switching commands is independent of the amount of available power, and therefore the two processes are simply interleaved.

\[
\text{FixedCapacityEPS}_2 = \\
\quad \text{let} \\
\quad \quad \text{CommandLogic} = \text{cmd}\_sw?x \rightarrow \text{power.load\_switch}!x \rightarrow \text{CommandLogic} \\
\quad \text{within} \\
\quad \quad \text{CommandLogic} || \text{FixedCapacityEPS}
\]

The power-switching behavior in the preceding example is very simple, and could in fact have been expressed directly as \(\text{LiftF}(\text{cmd}\_sw, \text{power.load\_switch}, fID)\). Many spacecraft, however, have somewhat more complicated switching behavior. For example, as fig. 5.27 and fig. 5.28 (excerpted from the *WIRE System Requirements* [24]) illustrate, the WIRE spacecraft includes both a power-switching action which occurs upon EPS startup, and switching of multiple loads simultaneously in response to both internal and external events. However, even complex switching requirements tend to involve little beyond translating events into switch actions. Thus, extending the behavior of the *CommandLogic* process to handle these additional requirements is generally a straightforward matter of adding external choice branches corresponding to each event which might cause a switch action, and adding sequential behavior components to model things like startup behaviors.

---

The power subsystem includes an isolation relay to disconnect the non-essential bus in the event that the spacecraft loads are drawing excessive current, the battery is in an undervoltage condition, or when the spacecraft goes into safehold.

---

Fig. 5.28: Power buses on the WIRE spacecraft.
One issue that does need to be addressed in extending the power-switching behavior to accommodate more complex requirements is the modeling of actions which can affect multiple subsystems simultaneously. For example, the previously mentioned WIRE EPS has the capability to switch off multiple loads at once. However, all of the switching events we have described so far involve a single subsystem at a time.

Modeling a simultaneous switch action in terms of individual subsystem events is problematic for two reasons: first, the use of individual switching events can result in unnecessary interleaving, which may cause an undesirable growth in the state-space of the system model; second, the actual switch action will most likely be implemented using a single relay, which means that the additional interleavings associated with using individual switching events misrepresent the way the system is actually intended to operate. To circumvent these difficulties, we define an auxiliary switching event, and require that all subsystem models which will be affected by the event respond both to their individual subsystem switching events, and to the new auxiliary event. This requirement can usually be fulfilled by applying the renaming operator, as we do in the following example. By ensuring that all of the subsystem models synchronize on the auxiliary switching event when they are composed into a system model, we can model simultaneous switching without incurring any unwanted interleaving.

**Example 5.4.4.5.** The CommandLogic2 process approximates the power-switching behavior of the WIRE EPS. Following the WIRE example, we split the power bus into an essential bus and a nonessential bus:

\[
\text{channel } \text{essential} \_\text{bus, nonessential} \_\text{bus : Power}
\]

We assume that the sets

\[
\text{Essential } \subseteq \text{Subsystem. OnOff}
\]
\[
\text{Nonessential } \subseteq \text{Subsystem. OnOff}
\]

have been defined, and specify the power-switching commands that apply to each bus.
To the existing $EPScmd$ datatype we add commands for triggering safehold powerdown of the nonessential bus, and for triggering pyrotechnics. The additional channels $deploy$ and $eps\_fault$ are respectively used to signal deployment events, and to abstractly model overcurrent and undervoltage conditions.

\[
\text{datatype } EPScmd = eps\_tlm.\text{OnOff} \\
\quad \mid \text{sw.Subsystem.}\text{OnOff} \\
\quad \mid \text{safehold_powerdown} \\
\quad \mid \text{pyro.Mechanism}
\]

channel $cmd : EPScmd$

channel $deploy : \text{Mechanism}$

channel $eps\_fault$

The $\text{CommandLogic}_2$ process itself is defined as

\[
\text{CommandLogic}_2 =
\]

\[
\quad \text{let}
\]

\[
\text{Init} = \text{essential\_bus.load\_switch.cdh.on} \\
\quad \to \text{essential\_bus.load\_switch.adcs.on} \to \text{SKIP}
\]

\[
\text{Operational} =
\]

\[
\quad (cmd.\text{sw}?x : \text{Essential} \to \text{essential\_bus.load\_switch!}x \to \text{Operational})
\]

\[
\text{Operational} =
\]

\[
\quad (cmd.\text{sw}?x : \text{Nonessential} \\
\quad \to \text{nonessential\_bus.load\_switch!}x \to \text{Operational})
\]

\[
\quad
dot
\]

\[
\quad (cmd.\text{pyro}?m \to \text{deploy!}m \to \text{Operational})
\]

\[
\text{Operational} =
\]

\[
\quad (cmd.\text{safehold\_powerdown} \\
\quad \to \text{nonessential\_bus.load\_switch.all\_off} \to \text{Operational})
\]


\[
(\text{eps\textunderscore fault} \rightarrow \text{nonessential\textunderscore bus}\textunderscore load\textunderscore switch\textunderscore all\textunderscore off} \rightarrow \text{Operational})
\]

within

\text{Init ; Operational}

The process \text{Init} completes before the rest of the EPS behavior, and models internally-triggered switch events which occur on EPS startup.

We model the capability of the EPS to shut off all power to loads on the nonessential bus by assuming that the \textit{Subsystem} datatype has been extended with the symbol \textit{all}, and that all of the subsystem models interfacing to the \textit{nonessential\textunderscore bus} channel respond both to switching commands involving their subsystem name, and to commands involving the \textit{all} symbol. The latter assumption is most easily accommodated through a renaming which causes each subsystem switching event to be mapped both to itself, and to the \textit{all} switching event. For example, a Payload subsystem model could have the following renaming applied:

\[
\text{Payload}[\text{nonessential\textunderscore bus}\textunderscore load\textunderscore switch}\textunderscore pl \leftarrow \text{nonessential\textunderscore bus}\textunderscore load\textunderscore switch}\textunderscore pl, \\
\text{nonessential\textunderscore bus}\textunderscore load\textunderscore switch}\textunderscore pl \leftarrow \text{nonessential\textunderscore bus}\textunderscore load\textunderscore switch}\textunderscore all]
\]

\section*{5.4.5 Communications}

Spacecraft communications subsystems typically have a fairly simple behavior. The role of the communications subsystem is essentially to transform internal spacecraft signals into radio-frequency signals, and vice versa. However, although performing these transformations may require sophisticated electronics, the details of how the transformations are accomplished is not a significant consideration for determining system behavior. Nor are the details of modulation schemes or error control coding relevant at the level of abstraction in which we are interested. Since the information content of the signals on each side of the transformation is nominally the same, the communications uplinks and downlinks can be treated as black boxes that simply move messages from a channel representing one kind of
signal to a channel representing a different kind of signal. The “messages” that are moved between channels can be either symbols representing discrete packets of data, or symbols that represent key transitions in a stream of data.

**Example 5.4.5.1.** We model the downlink portion of a communications subsystem as a process that, when in a powered state, transforms messages on the `systembus` channel, which represents an internal spacecraft signal, into messages on the `downlink` channel, which represents a radio-frequency signal.

```plaintext
datatype SysBus = dl_cmd.DownlinkMsg | ···
channel systembus : SysBus
channel downlink : DownlinkMsg
channel power : Power

Downlink =

  let
  DL(s) =
    s = off & systembus.dl_cmd?m → DL(s)
  □
    s = on & systembus.dl_cmd?m → downlink!m → DL(s)
  □
    power.load_switch.dl?s' → DL(s')

  within
  DL(off)
```

The `Downlink` process in the preceding example effectively provides single-message buffering for downlinked messages, since any process that generates a message on the `systembus` channel is free to proceed, while the `Downlink` process blocks until the occurrence of the `downlink` event which signals actual message transmission. If desired, multiple-message buffering can be added using the buffer processes described in section 4.3.1.
We may sometimes wish to create a model in which no message buffering occurs, for example to represent a command uplink which immediately passes received commands to the CDH subsystem for execution. This requires a slightly different approach to modeling the transformation from one type of signal to another, since instead of using two separate events to represent the transformation we must somehow use a single event to simultaneously signify the presence of a message on two different channels. This feat can be accomplished by using the renaming operator to identify internal spacecraft events as radio-frequency events from the perspective of an external transmitter or receiver. The role of the process which represents the communications subsystem then becomes simply that of a “gatekeeper,” either permitting or preventing the occurrence of a communications event depending on the state of the subsystem.

**Example 5.4.5.2.** To model immediate execution of uplinked spacecraft commands we construct an *Uplink* process which acts as a gatekeeper for the occurrence of the *cmdin* events that signify receipt of a command by the CDH subsystem. We assume that commands are received from an external source via the *uplink* channel.

```
channel cmdin, uplink : SpacecraftCommand
channel power : Power

Uplink =

let

UL(s) =

  s = off & uplink?m → UL(s)
  □

  s = on & cmdin?m → UL(s)
  □

  power.load_switch.ul?s′ → UL(s′)

within

UL(off)
```
To use the *Uplink* process, we place it in parallel with a process *CDH*, which is assumed to represent the spacecraft CDH subsystem. The *Uplink* and *CDH* processes synchronize on *cmdin* events. We apply a renaming to the composite process which makes *cmdin* events appear as *uplink* events from the perspective of processes outside the renaming:

\[
(Uplink \parallel \{cmdin\} \parallel CDH)[cmdin \leftarrow uplink]
\]

Now, when the *Uplink* process is in the *off* state the composite process directly accepts *uplink* events, but does nothing with them since they are simply discarded by the *Uplink*. When the *Uplink* process is in the *on* state the composite process also accepts *uplink* events, but in this case the accepted events are actually renamed *cmdin* events, upon which the *CDH* process synchronizes in order to receive the command message. Thus, the *Uplink* process acts as a gatekeeper that only allows the *CDH* process to synchronize on command message events when the *Uplink* is powered on.

The basic communications subsystem behaviors that we have just described can be extended in a few different ways. A more complex communications subsystem model might include:

- Several different modes, each representing transmission or reception on different bands (e.g. S-band vs. X-band) in terms of differing output or input channels. Transitions between modes might occur upon command, or as the result of other spacecraft events.

- Subsystem mode power behavior, if communications subsystem load management is a key part of the system behavior.

- Fault events, and associated fault-tolerance behaviors.

However, modeling of all of these extensions to the basic communications behavior can be carried out using the same techniques as were discussed in section 5.4.2, so we will not consider them further here.
5.4.6 Payload

The payload subsystem is the primary contributor to achieving the spacecraft mission, and ultimately the reason for designing a spacecraft. Spacecraft missions can range from providing a communications relay or a navigation signal, through various kinds of observational and data collection missions in pursuit of scientific or military goals [19], to proposed ideas for missions such as on-orbit refueling or repair of existing satellites [125]. Consequently, the type of payloads that make up the payload subsystem also varies widely from spacecraft to spacecraft. However, as with other subsystems, modeling the behavior of a payload subsystem in CSP involves abstracting from the internal details of how the payload accomplishes its tasks, and focusing on defining the payload interactions at its command, data, and power interfaces.

Even with the application of abstraction, payloads for different missions often have widely differing expected behaviors. Some payloads may require little more as an input than electrical power, and simply produce a steady stream of data. Other payloads, as the excerpt from the WIRE Instrument to Spacecraft Computer System ICD [126] shown in fig. 5.29 demonstrates, may require careful coordination with other subsystems in order to successfully complete their tasks. As a result, unlike other spacecraft subsystems, it is much more difficult to develop a general approach for creating payload behavior models. Instead, we outline the basic questions that need to be addressed in developing a payload process model.

\[
\text{The WIRE C&DH will use three commands to perform all of a WIRE observation segment; the first is a command to the ACS to specify observation location and dither pattern; the second is to the WIRE instrument controller to initiate the observation; the third is to the WIRE instrument controller and will be executed about 20 seconds before the end of an observation segment and will perform whatever finish/stim flash operations are required.}
\]

Fig. 5.29: Payload operation requirements for the WIRE CDH.
The questions that should be considered in developing a payload process model are:

• **Inputs**

  – Does the payload depend on commands received from other subsystems? What effect do these commands have?
  
  – Does the payload behavior depend on data received from other subsystems? How does this data alter or affect the behavior of the payload?

• **Outputs**

  – Does the payload produce any data? If so, how is the data best represented in the process model: as discrete messages or as a stream?
  
  – For a message-based output, what triggers the production of a discrete message: external events, or decisions internal to the payload?
  
  – For a stream-based output, what are the qualitative values that an output data stream can communicate, and what controls the current value of the stream? If the stream is dependent upon a state, then a *StateTelemetryStream* process can be applied to model the stream.
  
  – Does the payload need to command other subsystems? Under what conditions are these commands generated?

• **Modes**

  – Does the payload have mode-dependent behavior? If so, a process which specifies the payload behavior in different modes, and the events that trigger transitions between different modes, such as that used for modeling the ADCS in example 5.4.2.3, would be a good starting point for the payload process model.
  
  – What is the power consumption in different modes? If the power consumption varies, then it may be helpful to incorporate a *SubsysModePower* process into the payload process model.
Example 5.4.6.1. As an example of working through the questions outlined above to develop a process model, consider the behavior of a notional still-frame camera payload for an observational mission:

- **Inputs** – We assume that, like the WIRE payload described in fig. 5.29, the camera must be commanded to make an observation. However, unlike the WIRE payload, we assume that each observation corresponds to the collection of a single image. Therefore, the observation can be adequately modeled as a single input event, and the camera does not require a command to terminate an observation. No data inputs are required.

- **Outputs** – Since observations are modeled as single events which occur upon command, the data output from the camera payload is most easily modeled in terms of discrete messages. We assume that each message contains image data in a bitmap format, and that communication of these messages immediately follows collection of an image.

- **Modes** – We assume that the camera has only two modes: *on* (powered) and *off* (unpowered). In the *off* mode the camera does nothing. In the *on* mode it responds to command inputs by making an observation and producing a data output. Transitions between the two modes are triggered by EPS switching events. The power consumption in each mode is defined by the mapping

  \[
  f_{\text{CameraModePower}} : \text{CameraMode} \rightarrow \text{Integer}
  \]
  \[
  f_{\text{CameraModePower}} = \{(\text{off}, 0), (\text{on}, 5)\}
  \]

  We model observation events using the channel *camera*. Since, from the description above, the actual data taken during an observation event has no impact on the behavior of the payload, we model it as a single abstract symbol at this point.
datatype CameraData = image
channel camera : CameraData

Based on the description above, we define the following datatypes and channels to represent the camera command and data interfaces. We also include a specification event channel to signal mode transitions. The power interface uses the same channel and type structure described in previous sections.

datatype CameraCmd = take_image
channel pl_cmd : CameraCmd
datatype CameraTlm = bitmap.CameraData
channel pl_tlm : CameraTlm
channel camera_modetrans : ModeTrans.OnOff

We construct the Payload process model from a parameterized process representing the behavior of the camera in each of its two modes, and a SubsysModePower process. The behavior of the camera in the on mode formalizes the description of the command/observation/output sequence described above.

Payload =

let

Camera(off) =
  (power.load_switch.payload.on → camera_modetrans.begin.on
   → camera_modetrans.end.on → Camera(on))
   □
pl_cmd.take_image → Camera(off)

Camera(on) =
  (power.load_switch.payload.off → camera_modetrans.begin.off
   → camera_modetrans.end.off → Camera(off))
5.4.7 Propulsion

The propulsion subsystem of a spacecraft provides the capability to exercise control over the orbit of the spacecraft, either changing to a new orbit, or maintaining an existing orbit in the face of external perturbations. Propulsion is sometimes also used to support spacecraft attitude control, either directly, by modifying the attitude, or indirectly, by providing a torque suitable for removing accumulated spacecraft angular momentum. Almost all spacecraft propulsion is carried out using some type of rocket, and it is on this form of propulsion that we focus our behavior modeling efforts.

In comparison to most of the other spacecraft subsystems, propulsion subsystems typically have a fairly simple behavior. Like the example in fig. 5.30, most propulsion subsystems consist of little more than thrusters, propellant tanks, and plumbing to carry propellant between the tanks and the thrusters. Control over the thrusters is exercised by opening and closing valves contained in the thrusters, and in the propellant lines. Our modeling of propulsion system behavior centers on the states of these valves, and how changes in the valve states affect the orbit state and the quantity of propellant remaining.

The elements of our approach to propulsion behavior modeling are the following:

- We model the propellant as an integer-valued quantitative resource (see section 4.3.1).
Fig. 5.30: Example schematic for a monopropellant blowdown propulsion subsystem.

- We model the orbit state as an abstracted set of qualitative symbols, similar to the way that attitude states were modeled in section 5.4.2.

- We abstract from individual thrusters, and consider the state of the propulsion subsystem as an aggregate of the underlying thruster states, which may either be providing thrust, or not. The effects of a given thruster burn are determined by abstract parameters carried in the events which signal the start and end of a burn, rather than in the actual choice of individual thrusters to fire.

- For simplicity, we assume that the $\Delta v$ required for a given maneuver is always the same, and that the propellant consumed to produce that $\Delta v$ also remains constant over the operational life of the spacecraft.

- We model the change in propellant level that results from a maneuver as a single $delta$ event applied to the $QuantResource$ that represents the propellant level. The $delta$ event occurs between the events that represent the beginning and end of the burn used to execute a maneuver.

The following example illustrates the preceding ideas in action.
Example 5.4.7.1. Consider a typical spacecraft propulsion system, capable of providing both orbit adjustments, and dumping of spacecraft momentum. These two different types of maneuvers are represented by the datatype

\[
\text{datatype } \text{BurnType} = \text{momentum\_dump} \mid \text{orbit\_maneuver}.\text{OrbitState}
\]

where the \text{OrbitState} type defines the qualitative orbit states the spacecraft may take on:

\[
\text{datatype } \text{OrbitState} = \text{launch\_orbit} \\
\mid \text{transfer\_orbit} \\
\mid \text{mission\_orbit} \\
\mid \text{disposal\_orbit}
\]

The explicit interface to the propulsion subsystem consists of commands to actuate latch valves, which enable and disable the thruster systems as a whole, and commands to start and stop thruster burns of a given type. The latter are an abstract representation of underlying concrete commands to actuate particular combinations of individual thruster valves.

\[
\text{datatype } \text{OpenClose} = \text{open} \mid \text{close} \\
\text{datatype } \text{PropCmd} = \text{latch\_valve}.\text{OpenClose} \\
\mid \text{start\_burn}.\text{BurnType} \\
\mid \text{stop\_burn}.\text{BurnType}
\]

channel \text{prop\_cmd} : \text{PropCmd}

The \text{orbit} channel provides an implicit interface between the propulsion model and models of other subsystems that depend on the orbit state.

channel \text{orbit} : \text{StateIF}.\text{OrbitState}
Internally, the propulsion subsystem process model includes a QuantResource process representing the available amount of propellant. The following channels define the specifications events that form the interface for this QuantResource process. The value MAXPROP represents the maximum (integer-scaled) quantity of available propellant.

\[ MAXPROP = 15 \]

channel \(prop\_delta, prop\_getval, prop\_trans: \{-MAXPROP..MAXPROP\}\]

The Prop process defines the behavior of the propulsion subsystem. Within this process, the Orbit and Propellant processes represent the orbit state and propellant level respectively. The DUMP DELTA constant and the \(f_{prop}\) function define the change in propellant level associated with each kind of maneuver. We assume that maneuvers in which the starting and final states are mission_orbit represent station-keeping maneuvers. Maneuvers that generate orbit state transitions not explicitly defined by \(f_{prop}\) are assumed to be errors, and generate a maximal change in propellant level. The Off, Standby, OrbitAdjust, and MomentumDump processes define the different modes of the propulsion subsystem.

\[ Prop = \]

let

\[
\begin{align*}
    Orbit &= \text{AssignableState}(\text{orbit.setval}, \text{orbit.getval}, \\
             & \hspace{1cm} \text{orbit.trans, launch_orbit}) \\
    Propellant &= \text{QuantResource}(\text{prop_delta}, \text{prop_getval}, \\
                               & \hspace{1cm} \text{prop_trans, 0, MAXPROP, MAXPROP}) \\
    DUMP\_DELTA &= -1 \\
    f_{\text{prop}}(\text{launch_orbit, transfer_orbit}) &= -5 \\
    f_{\text{prop}}(\text{transfer_orbit, mission_orbit}) &= -2 \\
    f_{\text{prop}}(\text{mission_orbit, mission_orbit}) &= -1 \\
    f_{\text{prop}}(\text{mission_orbit, disposal_orbit}) &= -5 \\
    f_{\text{prop}}(_, _) &= -MAXPROP
\end{align*}
\]
The basic modeling approach just outlined can easily be extended with additional commands and subsystem modes to model design-specific propulsion components, such as the catalyst-bed heaters found in many mono-propellant systems. We can also use the modeling techniques described earlier in this chapter to incorporate things like fault events and fault tolerance measures, and load_delta events to represent variations in power consumption in different propulsion modes.
5.4.8 Structures and Mechanisms

Spacecraft structures and mechanisms are often grouped into a single subsystem [9,19], which provides both mechanical support for the rest of the spacecraft subsystems, and motion for those subsystems which require it. Spacecraft structures do not generally exhibit behavior, at least in the sense that we are using the term in this dissertation. Low-cyclic spacecraft mechanisms provide deployment of spacecraft components that must be stowed for geometric or environmental reasons, such as deployable solar arrays or antennas. High-cyclic mechanisms provide articulation for components that are required to move relative to the rest of the spacecraft to perform their function, such as sun-tracking solar arrays [19]. The behavior associated with both kinds of mechanisms is straightforward, and can be modeled in terms of commanded changes in qualitative states.

The states for deployment mechanisms are essentially binary: either the actuated component is deployed, or it is not. Modeling the behavior of such a mechanism is a simple matter of changing the deployment state based on received commands.

The states used to represent articulation mechanisms are typically more complex than those for deployment mechanisms, since it may be necessary to represent key changes in the position or speed of the articulated component. The typical behavior exhibited by articulation mechanisms is well-suited to a modeling approach essentially the same as that used to model attitude control in section 5.4.2, in which commands are translated into changes in the value of the qualitative state.

We model the effect of both deployment and articulation mechanisms, on the spacecraft components they are intended to control, in terms of implicit interfaces tied to the state of the mechanism. Although we consider “mechanisms” as a separate subsystem, it is usually more convenient to include a mechanism process model as part of the larger process model for the subsystem to which the mechanism provides service, rather than grouping all of the mechanism process models as an explicit subsystem process of their own. This approach has the advantage of making obvious the association between each mechanism and the subsystem to which that mechanism provides services.
Example 5.4.8.1. An instrument payload for a scientific spacecraft relies on a deployable antenna for its measurements. The antenna is a two-piece component, and both pieces must be deployed in order for samples taken using the antenna to provide meaningful data.

In this case, we model the deployment state of the mechanism as a set containing the identifiers of the deployed antenna elements. Reception of a `deploy.ant` command by the deployment mechanism results in the identifier included as part of the command being added to the deployed set.

\[
\text{datatype } \text{DeploymentCmd} = \text{ant.}\{1..2\}
\]
\[
\text{channel } \text{deploy} : \text{DeploymentCmd}
\]

Reports of a successful deployment are communicated to other subsystems through the payload telemetry interface.

\[
\text{datatype } \text{PLtlm} = \text{tlm}_\text{deploy.}\{1..2\} | \cdots
\]
\[
\text{datatype } \text{SysBus} = \text{pl_tlm.PLtlm} | \cdots
\]
\[
\text{channel } \text{systembus} : \text{SysBus}
\]

The Payload process model includes the process model for the deployable antenna. The interaction between the antenna and the instrument it supports take place via the implicit interface \textit{antenna}. When the antenna has been fully deployed, the antenna process permits meaningful sample data to be taken by the instrument. Otherwise, the instrument receives only junk data. We leave out the process model for the instrument, since it isn’t relevant to defining the mechanism model.

\[
\text{datatype } \text{ScienceData} = \text{sample} | \text{junk}
\]
\[
\text{channel } \text{antenna} : \text{ScienceData}
\]
\[\text{Payload} = \]
\[
\text{let}
\]
\[
\text{Instrument} = [\text{Instrument process model}]
\]
\[
\text{Antenna}(\text{deployed}) =
\]
\[
(\text{deploy.ant}?x \rightarrow \text{systembus.pl_tlm.tlm_deploy!}x \rightarrow \text{Antenna}(\text{deployed} \cup \{x\}))
\]
\[
\text{ankan!sample} \rightarrow \text{Antenna}(\text{deployed})
\]
\[
\text{ankan!junk} \rightarrow \text{Antenna}(\text{deployed})
\]
\[
\text{within}
\]
\[
(\text{Instrument} \parallel [\text{kan}] \parallel \text{Antenna}(\emptyset))
\]

As with other subsystem behavior modeling approaches, the techniques described here for modeling mechanism behavior can readily be extended to incorporate additional behavior, such as faults and power consumption variations.

5.4.9 Thermal Control

The thermal control subsystems of most spacecraft designs are primarily passive. That is, thermal regulation of the spacecraft is achieved by applying a combination of surface finishes, radiators, and insulation to modulate heat absorption and rejection. There is no behavior, in the sense that we use the term, associated with these approaches to thermal control, and thus no need to develop a thermal control process model. However, some spacecraft thermal control subsystems make use of active thermal control, in the form of things like heaters, pumped fluid loops, and louvers. These active thermal controllers are typically operated in a closed-loop manner, and do have some behavior associated with them, although it tends to be fairly straightforward behavior.

When an active thermal control subsystem is operating correctly, the system-level effects of the controller appear primarily as variations in power consumption. In a process
model of an active thermal control subsystem, changes in the thermal control power consumption may be triggered by external events (e.g. the occurrence of eclipse), as well as implicit interface events or specification events (e.g. mode transitions) which signal changes in the thermal state of some other subsystem. A thermal control failure may induce a corresponding fault event in a thermally sensitive subsystem.

**Example 5.4.9.1.** A simple active thermal control subsystem has a power consumption which varies depending on whether or not the spacecraft is presently in sunlight, or in eclipse. The behavior of this subsystem can be modeled as a mode-based control system similar to the ADCS models of section 5.4.2, except that instead of translating commands into attitudes, the thermal control behavior translates thermal events (the transitions to and from eclipse in this case) into load_delta events.

```plaintext
datatype SunState = in_sun | eclipse
channel sun_state = StateIF.SunState
ThermalControl =

let
  Controller(off) = power.load_switch.thermal.on →
  ((sun_state.getval.in_sun
    → power.load_delta.thermal.5 → Controller(in_sun))
  □
  (sun_state.getval.eclipsed
    → power.load_delta.thermal.10 → Controller(eclipsed)))

Controller(in_sun) =
  power.load_switch.off → power.load_delta.thermal.−5 → Controller(off)
  □
  sun_state.trans.eclipse → power.load_delta.thermal.5 → Controller(eclipsed)
```
Controller(eclipsed) =

\[(power\_load\_switch\_off \rightarrow power\_load\_delta\_thermal. - 10 \rightarrow Controller(\text{off}))\]

\[\square\]

\[(sun\_state\_trans\_in\_sun \rightarrow power\_load\_delta\_thermal. - 5 \rightarrow Controller(\text{in\_sun}))\]

within

Controller(\text{off})

5.5 System Analysis

Subsystem process models developed using the approaches described in the previous section are useful for precisely defining the behavior of individual subsystems. However, the real value of such models lies in being able to analyze the behavior that emerges when the subsystem models are connected together, and interact with one another. In this section, we describe how subsystem models may be combined into a system model suitable for analysis. More importantly, we look at how the resulting system models can be analyzed to verify that the system possesses desirable properties. We focus on checks that can be automated using FDR. Appendix D contains a machine-readable CSP example of a spacecraft behavior model built using the approach developed in this chapter, and several examples of property verification on this model.

5.5.1 Connecting Subsystems

The connections between subsystems can be as critical to defining overall spacecraft system behavior as the behavior of the individual subsystems. Obviously, missing connections between subsystems, such as the failure to connect a subsystem command input to the command bus, will produce a non-functional spacecraft. But, in addition, the configuration of the connections between subsystems can be an important design consideration. For example, the choice between allowing individual subsystems to communicate directly with one
another, or requiring all inter-subsystem communication to be routed through some central
executive, may have important implications for the management of overall spacecraft state
and the robustness of the spacecraft to single-point failures.

Given a selection of subsystem behavior models, part of the spacecraft system design
process may involve experimenting with different configurations of subsystem interconnec-
tions, and exploring the resulting behavior of the composite system. Even if this is not the
case, providing an explicit definition of how the spacecraft subsystems are connected to form
the composite system is a key part of defining the spacecraft system, as illustrated by the
prevalence of system block diagrams in spacecraft design documentation, and a necessary
prerequisite to the analysis of a composite system model.

Although any of the synchronizing parallel operators provided by CSP could probably
be applied to define the composition of spacecraft subsystem process models into a system
model, our approach relies on the use of the generalized alphabetized parallel operator.
Unlike the binary parallel operators, building a multi-process composition using the gener-
alized alphabetized parallel operator does not require consideration of the order in which
the subsystems are composed when defining the interface set for each subsystem process.
Instead, using the generalized alphabetized parallel operator, we simply define an interface
set for each subsystem which specifies exactly those channels over which the subsystem is
willing to communicate. Connections between a given pair of subsystems are then defined
by the existence of a non-empty intersection between the interface sets of the two subsystem
process models. This aspect of the generalized alphabetized parallel operator makes it an
appealing choice for defining subsystem composition, since it means that subsystem process
models can be added to, modified, and removed from, the composite system model without
requiring any changes in the interface sets of other subsystem process models.

**Definition 38** (Spacecraft System Model). A *spacecraft system model* is a composite pro-
cess of the form

$$SysModel = \| (IF, Subsys) : Subsystems \bullet IF \circ Subsys$$
where $Subsystems$ is a set of 2-tuples, with each 2-tuple consisting of a subsystem process, $Subsys$, and a corresponding interface set, $IF \subseteq \alpha(Subsys)$. □

Due to the semantics of the alphabetized parallel operator used in definition 38, the interface set for each subsystem must include any specification events that we want to be visible at the level of the composite system model. Events which we do not want to be visible must be hidden within the subsystem process model, since if they are not hidden, and do not appear in the interface set, they will be blocked from executing.

As a consequence of using synchronization on common channels to define subsystem connectivity, the connections between subsystems are, to a certain extent, governed by the choice of channels that we use in building the models of each subsystem. However, while all of the examples we have presented in this chapter embed the use of specific channels within the behavior description, there is nothing to prevent us from defining subsystems models such that they are parameterized by their interface channels. Even in the case of process models that include embedded channel names, the CSP renaming operator can be used to redefine the names of these channels at the interface between the subsystem model and the rest of the system. As a result the definition of subsystem connectivity ultimately resides in the the interface sets which specify how the subsystems process models synchronize with one another.

Example 5.5.1.1. Assume that we have defined a collection of spacecraft subsystem process models, consisting of the processes $ADCS$, $CDH$, $Comm$, $EPS$, and $Payload$. To define a composite spacecraft system model, we first define the interface sets for each subsystem.

\[
ADCS_{IF} = \{power.load_switch.adcs, power.load_delta.adcs, \\
\text{systembus.adcs_cmd, systembus.adcs_tlm,} \\
attitude.trans, fault\}\n\]

\[
CDH_{IF} = \{power.load_switch.cdh, cmdin, systembus\}\n\]

\[
Comm_{IF} = \{cmdin, power.load_switch.ul, downlink, \\
\text{systembus.dl_cmd, power.load_switch.dl}\}\n\]
For most of the subsystems, the interface set simply consists of the appropriate command, telemetry, and power channels. However, a few of the subsystems also include specification events intended to be visible at the system level. Thus, for example, the $EPS_{IF}$ interface set includes the exception-events $qr\_exception$, and $eps\_exception$, since we want to be able to verify at the system level that they do not occur. However, $EPS_{IF}$ does not include the events which are internally used to track the levels of allocated and available power within the EPS, and as a consequence the definition of the EPS process model will need to hide these events.

Given the interface sets defined above, the composite spacecraft system model is

$$Subsystems = \{(CDH_{IF}, CDH),$$
$$\quad (EPS_{IF}, EPS),$$
$$\quad (ADCS_{IF}, ADCS),$$
$$\quad (Payload_{IF}, Payload),$$
$$\quad (Comm_{IF}, Comm)\}$$

$$SCsys = \| (IF, Subsys) : Subsystems \bullet IF \circ Subsys[cmdin \leftarrow uplink]$$

Note that this composite model includes a renaming that maps the $cmdin$ channel to the $uplink$ channel, as described in sec. 5.4.5.

Figure 5.31 illustrates the connections between subsystems defined by the preceding CSP expressions.
5.5.2 Verifying System Behavior

Verification involves determining whether or not a given design is, in some sense, *correct*. The correctness of a design is not an absolute, but rather is relative to the requirements the design seeks to fulfill. Correctness is decided by analyzing the design to ensure that it possesses certain desirable properties. Classical conceptual-level spacecraft design and analysis [19] focuses on properties such as positive launch mass margin, i.e. the spacecraft mass is less than the launch mass capability of the intended launch vehicle, and positive power margin, i.e. the spacecraft EPS is capable of producing more power than is consumed by the other subsystems. These properties are verified by, for example, summing simplified, abstracted models of the mass of individual subsystems to find a total system mass, and comparing it against the required maximum spacecraft system mass. In the similar way, we seek to verify that the behavior of the spacecraft is correct by “summing” simplified, abstracted models of the behavior of individual subsystems into a system behavior model, and comparing the resulting system behavior against various required behavioral properties.
In this section we consider several different kinds of system behavior properties which can be expressed in CSP, and automatically verified using the FDR refinement checker. In defining these properties, we follow the nomenclature developed by Avizienis *et al.* for describing different characteristics of dependable systems [127], using the term *error* to refer to an internal problem within a system, and *service failure* (which is not the same as a *failure* in the CSP process-semantics sense) to refer to an observable deviation from the behavior required of the system. Since all of the examples in this section are intended to be representative of checks that can be performed using FDR, we use the machine-readable CSP$_M$ syntax throughout the section. The sample spacecraft system model in appendix D includes a complete system model of a spacecraft against which example properties of the kinds defined in this section can be verified.

### 5.5.3 Model Sanity

The most straightforward properties that we might verify for a given spacecraft system model are basic sanity checks that ensure the model is producing meaningful results. In classical spacecraft conceptual design, model sanity can be checked by, for example, verifying that the proportions of mass allocated to the different subsystems are not wildly outside of the historical range. In the realm of spacecraft behavior design, we can check model sanity by verifying that the modeled spacecraft is able to operate at all (i.e. that it is free of halt-failures), regardless of whether or not those operations are correct. In particular, we can verify that the spacecraft model has the properties *freedom from livelock*, and *freedom from deadlock*.

Freedom from livelock indicates that the spacecraft never reaches a state in which the subsystems continue to communicate with one another, but the system no longer produces observable outputs or responds to external inputs. A spacecraft which continues to maintain its attitude, and to collect data, but which never accepts commands from the ground station, and never transmits data, can be considered to be in a livelocked state. While a spacecraft in a livelocked state is, in some sense, still operating, it is effectively non-operational from an external perspective.
Example 5.5.3.1. The example in appendix D includes a check for freedom from livelock, expressed as an FDR assertion that the $SC_{system'}$ process does not diverge.

\[
\text{assert } SC_{system'} \cup \text{union}(\text{Faults, } \{|\text{deploy}|\}) : [\text{divergence free}]\]

Some explanation of the preceding assert statement is in order. The definition of $SC_{system'}$ hides internal spacecraft channels such as system and power buses. The channels for uplink, downlink, and launch vehicle separation remain externally observable. To aid the verification of other properties, the channel deploy, which signals mechanical deployments, and the channels

\[
\text{Faults} = \{|\text{fault, qr\textunderscore exception, eps\textunderscore exception}|\}
\]

which represent different faults and errors are also left observable. However, for the livelock-freedom check we hide the fault and deployment channels. As a result, verifying that the spacecraft model is free of livelock ensures that the spacecraft will always eventually perform some kind of interaction via the uplink, downlink, and separation channels. Furthermore, since the separation event can only occur once (as part of EPS initialization) the livelock-freedom check ensures that the spacecraft will always eventually either accept an uplinked command, or provide some kind of downlink signal.

Freedom from deadlock indicates that the spacecraft never reaches a state in which it is “stuck,” or no longer operating. A successful check for deadlock-freedom ensures that none of the subsystems make unwarranted assumptions about which communications to expect from the other subsystems. A deadlock situation might occur when, for example, a particular subsystem does not perform a deployment that is required for the rest of the mission to proceed, leaving all of the other subsystems waiting for a signal indicating a successful deployment.

Example 5.5.3.2. The example in appendix D includes a check for freedom from deadlock, expressed as an assertion that the $SC_{system}$ process failures refines the process $DF_{tick}$.

\[
\text{assert } DF_{tick} [F= (SC_{system} \ |\text{Faults}| \ \text{CHAOS}(\text{Faults})) \ \text{\textbackslash} \ \text{Faults})]
\]
There are two points to note about the preceding assertion. The first point is that we perform lazy abstraction of events in the set \texttt{Faults} prior to carrying out the refinement check, which ensures that deadlock-freedom of the \texttt{SCsystem} process does not rely on the occurrence of fault events – a spacecraft that \textit{requires} faults to occur in order to operate is obviously undesirable. The second point is that, for the same reasons outlined in example 4.3.4.2, we use the \texttt{DFtick} process to test deadlock-freedom, rather than FDR’s built-in deadlock check.

As with the sanity checks used in classical conceptual design, there is no requirement that a spacecraft \textit{must} pass the sanity checks. Rather, a failure to pass these checks may be an indication that the spacecraft is being called upon to do something unusual. However, designers should make sure that they understand why the check in question failed, and be able to elucidate the justification for proceeding with the design in the face of such a failure.

\subsection{5.5.4 Avoiding Bad Behavior}

The model sanity checks just discussed are useful for ensuring that a given spacecraft system model describes a spacecraft that does not unintentionally cease to operate. However, the sanity checks say nothing about whether or not the operations the spacecraft performs actually conform to the requirements of a particular mission. We consider here the problem of ensuring that the spacecraft never behaves in a way that is forbidden by its requirements. Verification of such a property is analogous to, for example, verifying that the intended spacecraft geometric configuration does not exceed the envelope of the launch vehicle fairing.

System properties that forbid certain actions are sometimes referred to as \textit{safety} properties. For spacecraft behavior verification, we distinguish two different kinds of safety properties. The first is concerned with \textit{errors} in internal spacecraft states, without regard to their impact on external behavior. The second kind of property focuses on externally observable \textit{service failures} involving forbidden behavior, without considering internal states. These two kinds of properties have slightly different uses. Verifying the first kind of property helps to ensure that the spacecraft design operates as intended, while verifying the
second kind of property ensures that the spacecraft fulfills its requirements. On first glance, the goals of verifying both kinds of properties may appear to be the same. However, it is entirely possible for a design to operate as intended, but for the intentions of the designer to not correctly fulfill the requirements. Similarly, a design may meet its requirements, but do so in an unintended manner, indicating that the designer does not fully understand the design. Verification of both kinds of properties is therefore useful.

Errors

In many cases, verifying that a spacecraft model is free of a particular class of error can be as simple as checking that the events which signal entry into a particular erroneous internal state can never occur. A check of this kind can be readily expressed in terms of a trace refinement assertion of the form

\[
\text{assert STOP } [T= \text{Sys} \setminus \text{diff(Events, Err)}]
\]

where \text{Sys} is a system model, and \text{Err} is a set of events that signal a particular class of errors. The assertion evaluates to true if none of the traces of \text{Sys} contain an event in \text{Err}.

Example 5.5.4.1. The \text{SCsystem} model in appendix D can be verified error-free with respect to overconsumption and underproduction of electrical power by checking the assertion:

\[
\text{assert STOP } [T= \\
\text{SCsystem} \setminus \text{diff(Events, \{|qr\_exception, eps\_exception|\})}]
\]

The simple assertion described above identifies any occurrence of a particular event as an error. However, some kinds of errors involve events which are acceptable in certain situations, but not in others. This is particularly true of errors that result from an interaction between subsystems, since the events produced by one subsystem may only be erroneous when another subsystem is in certain states. In these cases, a more complex process specification which defines the acceptable and unacceptable sequences of events must be developed. The form of the specification depends heavily on which sequences of events
are considered acceptable, although generic specifications such as the \textit{Between} and \textit{Outside} constraint processes defined in the previous chapter often provide a good starting point.

\textbf{Example 5.5.4.2.} To verify that the \texttt{SCsystem} model contained in appendix D only permits the science instrument to be used when the spacecraft is in the science attitude, we make use of the following assertion:

\begin{verbatim}
  GoodAtt = \{attitude.trans.earth_limb_scan\}
  BadAtt = diff({\{attitude.trans\}}, GoodAtt)
  assert Between(GoodAtt, BadAtt, {\{instrument.sample\}}) [T=
                           \texttt{SCsystem} \ diff(Events, {\{attitude.trans, instrument\}})
\end{verbatim}

The assertion evaluates to \texttt{true} if \texttt{instrument.sample} events, which indicate instrument use, only occur when the spacecraft is in the \texttt{earth_limb_scan} attitude. \hfill \blacksquare

\textbf{Failures of Commission}

By \textit{failures of commission}, we mean spacecraft service failures which involve the occurrence of a forbidden sequence of events at the interface between the spacecraft and its end-users. Typically, the interface in question involves channels representing uplinks, downlinks, and interfaces with the launch vehicle. Verifying the absence of failures of commission is different from verifying the absence of errors, in that the former says nothing about how the spacecraft internally accomplishes the avoidance of those failures. However, defining checks for failures of commission allows end-user expectations about the behavior of the spacecraft to be directly captured as part of the formal modeling process, and ensures that, from the perspective of a user, the spacecraft never does anything unexpected.

As with checks for freedom from errors, simple checks for freedom from failures of commission, i.e. checks for properties that involve placing an absolute ban on certain events, can easily be expressed using the same assertion form that we used for verifying simple error-freedom properties. However, more complex limitations on permissible behavior again require the development of property-specific assertions. The following two examples illustrate both kinds of check.
Example 5.5.4.3. Appendix D includes an example of a simple check to verify that the only science data transmitted over the downlink is data collected after the instrument antennas have been deployed, and while the spacecraft is in the required attitude for science. Acceptable science data transmissions are defined as the set of *missiondata* messages consisting of a *sample* (rather than *junk*) taken in the *earth_limb_scan* attitude:

\[
\text{GoodScienceOutput} = \\
\{\text{downlink.missiondata.formatted.}(m,a) \mid m \leftarrow \{\text{sample}\}, \\
a \leftarrow \{\text{earth_limb_scan}\}\}
\]

The property to be checked is expressed as an assertion that no trace of the *SCsystem* process contains any *missiondata* message not contained in the *GoodScienceOutput* set.

\[
\text{BadScienceOutput} = \text{diff}(\{|\text{downlink.missiondata}|\}, \text{GoodScienceOutput})
\]

\[
\text{assert STOP \{T= SCsystem \ diff(Events, BadScienceOutput)}
\]

Note that successful verification of this property does not guarantee that the spacecraft will not collect science data in attitudes other than the *earth_limb_scan* attitude. It only guarantees that if any data is collected in other, undesirable attitudes, that data will not be downlinked to the ground station.

Example 5.5.4.4. An example of a slightly more complex verification of freedom from failures of commission is a check to ensure that the *SCsystem* process never attempts to make downlink transmissions before the spacecraft has separated from the launch vehicle. The property in question is not an absolute ban on downlink events – a spacecraft which obeyed such a ban would be less than useful – but rather a ban on those events which lasts up until the occurrence of the *separation* event. The following assertion captures this requirement:

\[
\text{SepPrecedesDownlink} = \text{separation} \rightarrow \text{RUN}(\{|\text{downlink}|\})
\]

\[
\text{assert SepPrecedesDownlink \{T=}\}
\]

\[
\text{SCsystem \ diff(Events, \{|\text{separation,downlink}|\})}
\]
5.5.5 Requiring Good Behavior

Checking safety properties ensures that the spacecraft doesn’t do anything that it should not. But safety properties cannot guarantee that the spacecraft will do those things it is supposed to do, or even that it will do anything at all. Just as a spacecraft launch mass requirement can be met by launching an inert block of metal, a required safety property can be provided by simply launching a spacecraft that does nothing. Obviously, neither of these spacecraft would be particularly useful. Thus, in addition to negatively defining what the spacecraft may do, it is also necessary to positively define what the spacecraft must do. Properties that specify behaviors which must occur are sometimes referred to as liveness properties. There are several different kinds of liveness properties that are relevant to spacecraft behavior. We consider three of those kinds of properties here: scenarios, mandatory internal states, and freedom from service failures of omission.

Scenarios

Scenarios are perhaps the simplest way to specify what a spacecraft must do. Designers typically have at least a nominal mission scenario in mind when developing a spacecraft design. The act of building a spacecraft system model can help to clarify the nominal mission scenario, and also to identify other possible operational scenarios. Although individual scenarios can be checked using a process exploration tool such as ProBE, verification that a spacecraft system model, especially one with many hidden internal states, is capable of operating in accordance with a given scenario can be performed in a much faster, easier, and more repeatable manner by testing the scenario using a check within FDR. Indeed, it may be useful to develop a suite of such checks, each defining a different intended operational scenario. The spacecraft model can then be automatically checked against every scenario it is expected to be capable of performing. The suite of checks can be re-run whenever the spacecraft design is changed, to ensure that the design change has not altered the spacecraft behavior such that it can no longer operate in the desired way.

We assume that an individual scenario can be described as a single, linear, finite sequence of events which describes an expected series of interactions between the spacecraft
and its end-users. Thus, an individual scenario can be modeled as an EventSeq process, as defined in sec. 4.2.1. Our approach to automatically verifying the capability to perform a scenario is essentially a form of must-testing [99]: we place the spacecraft model in parallel with a process representing the scenario to be verified, and follow successful termination of the scenario event sequence with a special event which denotes successful completion of the test. We use a failures/divergences refinement assertion of the form

\[
\text{assert } (\text{success } \rightarrow \text{STOP}) \ [\text{FD} = \\
\quad (\text{Sys } [\text{!ScenIF}] \ (\text{EventSeq(S)}; \text{success } \rightarrow \text{STOP})))
\]

\[
\quad \backslash \text{diff(Events, \{success\})}
\]

to check that the model Sys is capable of performing the scenario defined by EventSeq(S). The refinement check will succeed if the composite process on the right-hand side of the expression is cannot refuse to produce a success event. A failed refinement check indicates that the model and scenario processes are able to deadlock or diverge prior to reaching the success event, and thus that the spacecraft is somehow able to refuse to provide the behavior expected of it.

**Example 5.5.5.1.** Appendix D contains a definition of a simple mission scenario, in which the spacecraft is initialized into its science data collection mode, and then downlinks a valid science sample. A diagram of the scenario appears in fig. 5.32. In CSP, the scenario is defined in terms of events on the separation, downlink, and uplink channels:

\[
\text{ScenIF} = \{\text{separation, downlink, uplink}\}
\]

\[
\text{Scenario} = \\
\text{EventSeq(<separation, downlink.modestatus.launch, downlink.modestatus.safe, uplink.mode.nominal, downlink.modestatus.nominal, uplink.att.science_attitude, downlink.missiondata.formatted.(sample, earth_limb_scan)>)}
\]
Because successful execution of the scenario is predicated on the assumption that no faults occur during the execution of the scenario, we suppress fault events by placing the SCsystem’ process in parallel with STOP, synchronizing on Faults. The resulting refinement assertion is:

\[
\text{assert (success -> STOP) [FD=} \\
((\text{Scenario; success }\rightarrow \text{ STOP}) \\
[\text{||ScenIF||}] \\
((\text{SCsystem’ [||Faults|| STOP}))) \\
\text{\ diff(Events, \{success\})}
\]

A check of the preceding assertion is successful, indicating that the spacecraft is capable of performing the specified scenario. Appendix D also includes an example of a refinement check for an invalid scenario, which omits the science attitude required by the spacecraft to initiate science data collection, and thus results in a failed check.
Mandatory Internal States

In sec. 5.5.4 we introduced the idea of defining safety properties applicable to the internal states of a spacecraft model. In addition to defining safety properties for internal spacecraft states, it is also useful to define liveness properties for these same states. Typically, what we wish to be able to specify is that a particular state or collection of states is not just reachable, but that under certain circumstances those states must be reached. That is, that the states in question are mandatory. Liveness properties of this sort are useful both for verifying that a system model is parsimonious, i.e. that it does not include unused states, and that our understanding of why the model reaches certain states is correct.

We formulate checks for mandatory internal states using an assumption/commitment specification style [128] in which the assumption encodes the conditions under which a given set of states is mandatory, and the commitment is that the mandatory internal states will occur. The generic form of an assumption/commitment assertion in CSP is

\[
\text{assert Commitment [FD=}
\]
\[
\text{(Sys \[|\text{AssumpIF}|\] Assumption) \ diff(Events, CommIF)}
\]

where AssumpIF is the alphabet of the Assumption process, and CommIF is the alphabet of the Commitment process. The assertion states that, given some assumption about the behavior of the environment in which Sys is operating, the behavior of Sys, when viewed in terms of just those events in CommIF, will not appear to be divergent, will not generate any traces forbidden by Commitment, and will not block any events required by Commitment. Example 5.5.3.2 includes a simple example of an assumption/commitment assertion, in which the assumption involves the occurrence of fault events, and the commitment is that the spacecraft be free of deadlock.

Example 5.5.5.2. Among the example liveness properties in appendix D is a refinement check which verifies that the spacecraft achieves all of the controlled attitude states into which it can be commanded. The controlled attitudes, and the corresponding attitude state transitions are:

\[
\text{ControlledAtt = \{sun_pointing, earth_limb_scan}\}
\]
AttTrans =

\{\text{attitude.trans.sun_pointing, attitude.trans.earth_limb_scan}\}

The assumption in this case involves the commands that may be sent to the spacecraft. Specifically, we assume that the spacecraft may be either left in its safe mode, or commanded into the nominal mission mode. We further assume that every attitude command is sent once, although the order in which the commands are sent is arbitrary. Because the interleaving operator is used to compose the two command assumptions, any combination of mode and attitude command is possible.

Assumption =

let

ModeCmdAssumption = ((uplink.mode.nominal -> STOP) |~| STOP)

AttCmdAssumption({}) = STOP

AttCmdAssumption(Cmds) =

|~| cmd:Cmds @ uplink.att.cmd

\rightarrow AttCmdAssumption(diff(Cmds,\{cmd\}))

within

ModeCmdAssumption ||| AttCmdAssumption(AttitudeCommand)

The commitment we choose to verify is that, given the assumptions outlined above, every controlled attitude is achieved at least once. This is effectively a check that all of the controlled attitudes are reachable.

Commitment =

let

AttCommit({}) = CHAOS(AttTrans)

AttCommit(Atts) =

|~| a:Atts @ attitude.trans.a \rightarrow AttCommit(diff(Atts,\{a\}))

within

AttCommit(ControlledAtt)
The refinement assertion follows the form described earlier:

\[
\text{assert Commitment [FD=}
\]

\[
\text{(SCsystem [\{\text{uplink}\}\}] Assumption)} \setminus \text{diff(Events, AttTrans)}
\]

Checking this assertion in FDR verifies that all controlled attitude states are indeed reachable. On the other hand, if the \text{f\_ACS} function in the ADCS portion of the spacecraft model is altered such that all of the attitude commands produce a \text{sun\_pointing} attitude, the spacecraft can never reach the \text{earth\_limb\_scan} attitude, and the refinement check will fail.

\section*{Failures of Omission}

Specifications associated with \text{failures of omission} are the liveness counterparts of failures of commission. We use the term failure of omission to mean a service failure which results from the spacecraft failing to perform some required behavior. Thus, a failure of omission involves a spacecraft failing to perform some required interaction with an end-user.

In some cases, the required behavior upon which a freedom from failures of omission check is based may be directly derived from the spacecraft requirements. In other cases, a collection of scenarios may suggest a more general behavior invariant of some sort, which can be captured as a check for freedom from failures of omission. In either situation, the refinement check defines some aspect of externally observable behavior which is required of the spacecraft, and the conditions under which that behavior is required. Given the preceding description, it should come as no surprise that the refinement assertion is well-suited to being defined using an assumption/commitment style. The following example illustrates the use of an assumption/commitment assertion to specify freedom from a failure of omission.

\section*{Example 5.5.5.3.} This example again draws from appendix D. We seek to verify that the spacecraft must return valid science data, under the following assumptions:

- The spacecraft initially receives the commands necessary to move it into an attitude and mode sufficient to allow science data collection to commence.
• The spacecraft consistently receives commands to start a new sample run, whenever such commands are needed.

• Upon detecting that the spacecraft has experienced a fault, the ground station transmits the command necessary to reenter a science mode.

• Faults occur nondeterministically. Only a finite number of faults can occur, i.e. the spacecraft cannot diverge on faults.

These assumptions are captured in the CmdAssumption and FaultHypothesis processes:

\[
\text{CmdAssumption} =
\]

\[
\text{let}
\]

\[
\text{RunSampling}(n) =
\]

\[
(n \leq 0) \& \text{uplink.mode.nominal} \rightarrow \text{RunSampling}(5)
\]

\[
[]
\]

\[
(n > 0) \& \text{downlink.missiondata?}_\rightarrow \text{RunSampling}(n-1)
\]

\[
[]
\]

\[
\text{downlink.fault_occurred} \rightarrow \text{CmdAssumption}
\]

\[
\text{within}
\]

\[
\text{uplink.mode.nominal}
\]

\[
\rightarrow \text{uplink.att.science_attitude}
\]

\[
\rightarrow \text{RunSampling}(5)
\]

\[
\text{FaultHypothesis} =
\]

\[
\text{let}
\]

\[
\text{FH}(0) = \text{STOP}
\]

\[
\text{FH}(n) = (\text{fault} \rightarrow \text{FH}(n-1)) \mid \text{STOP}
\]

\[
\text{within}
\]

\[
\text{FH}(3)
\]
The commitment in this case can simply be represented by the process $DF(A)$, which provides non-terminating deadlock-free behavior over the set of events $A$. The refinement assertion to verify that the spacecraft successfully provides valid science data when placed in the correct mode for data collection is then

$$\text{assert } DF(\text{GoodScienceOutput}) \text{ [FD=}
$$

$$((\text{SCsystem'})
$$

$$\text{[|{|uplink,downlink.fault_occurred,downlink.missiondata|}|]}
$$

$$\text{CmdAssumption})
$$

$$\text{[|{fault}|]} \text{ FaultHypothesis}
$$

$$\text{\ diff(Events, GoodScienceOutput)}
$$

Checking this assertion with FDR confirms that the spacecraft system model defined in appendix D successfully produces valid science data.

5.5.6 Verification Against a System Behavior Specification

In addition to verifying individual properties, it is also possible to check a system model against a system behavior specification of the sort described in the previous chapter. Given a system behavior specification, it is thus possible to directly verify that the proposed spacecraft design actually implements the specified behavior. Incompatibilities between the modeled design and the specification may indicate design errors, or they may elucidate areas in which the specification should be more precise. In either case, verifying a system model against a specification helps to clarify the designers’ understanding of both the specified and designed behaviors, and can increase confidence that the design will provide the behavior required of it.

Verification against a system behavior specification can typically be expressed using a straightforward refinement assertion of the form

$$\text{assert } SC\text{spec } [M= SC\text{system}$$
In some cases, it may also be necessary to include within the refinement assertion processes that define any assumptions about either the environment in which the spacecraft is to operate, or the occurrence of specification events within the spacecraft:

\[
\text{assert } \text{SCspec} \[|\text{AssumpIF}|\] \text{Assumption} \[M=\text{SCsystem} \[|\text{AssumpIF}|\] \text{Assumption}
\]

For example, the behavior of the spacecraft may rely on an assumption about how the ground station will command the spacecraft. Incompatibilities between the specification and model outside of this assumed environment is likely to be irrelevant, since any failure of the system in this situation would be a consequence of a malfunctioning ground station rather than an incorrect spacecraft design.

**Example 5.5.6.1.** Appendix D includes a revised version of the system behavior specification first presented in the previous chapter, and several refinement assertions which check the spacecraft system model against this revised specification. All of these assertions make use of the SCsystem' process, which hides the internal channels of the system model.

The traces refinement assertion provides a guarantee that the spacecraft system model does not do anything forbidden by the specification.

\[
\text{assert } \text{SCspec} \[T=\text{SCsystem}'\]
\]

The two failures refinement assertions provide guarantees that the spacecraft does the things required by the system specification.

\[
\text{assert } \text{SCspec} \[F=\text{SCsystem}'\]
\quad \text{assert } \text{SCspec} \[|\text{Faults}|\] \text{CHAOS(Faults)} \[F=\text{SCsystem}' \[|\text{Faults}|\] \text{CHAOS(Faults)}
\]

Failures refinement alone is sufficient for these assertions, because the SCsystem' is already known to be free of divergence (see example 5.5.3.1). The second of the two assertions is an example of the inclusion of an assumption process, in this case regulating the occurrence of fault events, as part of the refinement assertion.
Early attempts to verify the SCsystem process against SCspec helped to uncover a number of problems with the design, including poor assumptions about when ADCS faults might be a problem, and a mistake in the definition of the CDH safe mode which failed to fully account for the handling of ADCS faults. As detailed in appendix D, the verification process also led to a number of revisions to the specification which clarify what the spacecraft is supposed to do.

5.5.7 Automated Verification in Action

The example spacecraft system model which appears in appendix D illustrates the use of a number of the modeling approaches developed in this chapter. The model is simple, but non-trivial. It is simple in the sense that the modeled spacecraft is intended to be a relatively simple-minded scientific spacecraft which relies on ground commands for most of its operations. It is non-trivial in that it includes elements such as deployments, streaming attitude telemetry, mode-based power for at least some of the subsystems, an EPS which provides varying power depending on the spacecraft attitude state, and a CDH capable of a limited amount of fault-handling.

In addition to the spacecraft system model, appendix D contains a number of example refinement assertions, including several which demonstrate that the system model refines a system behavior specification. Using FDR (see fig. 5.33) to verify all of the example refinement assertions that appear in appendix D takes around 2 minutes 40 seconds, running on a 1.5 GHz PowerPC. Due to the way in which FDR operates on processes which appear on the left-hand side of a refinement assertion, the checks which include more complex processes on the left-hand side consume the most time. For example, checking the assertion

\[
\text{assert SCspec } \left[ |\text{Faults}| \right] \text{ CHAOS(Faults)} \ [F= \text{SCsystem' } \left[ |\text{Faults}| \right] \text{ CHAOS(Faults)}
\]

takes approximately 35 seconds, during which time FDR explores a total of 613,074 states, and 2,639,867 transitions.
Detecting Design Errors

Although the present version of the model which appears in appendix D successfully passes all of the refinement checks included in the appendix, it did not always do so. Indeed, a large part of the value of developing process algebraic models of the spacecraft system is the fact that such models can be analyzed to uncover and correct design errors.

Example 5.5.7.1. Despite being intentionally designed to avoid overconsumption of electrical power, an early revision of the SCsystem model, when checked with the assertion

\[
\text{assert STOP [T=} \\
\text{SCsystem \ \text{diff}(Events, \{|qr\_exception, eps\_exception|\})}
\]

was found to produce resource exception-events which indicated that the power system could become overloaded.

By analyzing the counterexample traces produced by FDR, the cause of the errors was eventually determined to be an improperly handled interaction between the CDH and ADCS subsystems. Specifically, the problem lay in a CDH initialization sequence that did not correctly wait for the attitude to transition out of the uncontrolled state before switching the payload on. Resolving this problem led to the introduction of the AwaitAttitudeAcq
portion of the CDH startup behavior. Analysis of the revised model confirmed that the modified CDH design did not induce EPS overloads.

In retrospect, the failure to include sensing of attitude transitions during CDH startup, as described in the preceding example, was an obvious omission. But the problem was overlooked in the initial formulation of the model, despite the fact that the model itself is relatively simple, and that the modeled design was deliberately intended to address the variability in EPS power production capability in different attitudes. Similar omissions could no doubt occur during the design of a real spacecraft, and, in the absence of analysis, might remain undetected until system integration and test, or even until the spacecraft was operating on-orbit. The ability to catch such errors as early as possible is one of the primary benefits of performing behavior analysis and verification.

**Scalability Considerations**

Although we have called the model in appendix D “non-trivial,” it is nevertheless likely that modeling a spacecraft with more complex mission behaviors than those assumed for our example model will result in larger state-spaces, and corresponding increases in the time it takes to complete a set of refinement checks. Had verifying our non-trivial example model involved run-times on the order of hours, rather than minutes, the prospects for checking more complex models would appear somewhat grim. However, since it is not uncommon for computational analyses of other aspects of a spacecraft design to take hours, or sometimes days, even significant increases over the computational time observed for verifying the example model should not preclude the inclusion of behavior modeling and analysis in the spacecraft design process.

Should the verification run-times for more complex behavior models become undesirably large, several avenues of mitigation are available. First of all, there are a number of techniques, such as hierarchical state-space compression [129] and watchdog transformations [130], which we have not applied to the example spacecraft model, but which can be exploited to make tractable the verification of process models with very large state-spaces.
Recent work on combining state-exploration and theorem-proving to perform refinement checks on systems having infinite state-spaces may also be applicable [69]. Other options for keeping the size of the model state-space in check include reducing interleaving by introducing system-wide “ticks” to synchronize different subsystems, and breaking the model into several less complex models, each representing the behavior of the spacecraft in a different phase of its mission.

5.6 Summary

The essential task of the spacecraft systems engineer is to ensure that the various subsystems which make up a spacecraft interact harmoniously, and that the resulting spacecraft system is capable of performing its mission. The existing approaches to reasoning about spacecraft subsystem interactions which are available to spacecraft engineers are for the most part informal and intuitive rather than analytical. Those tools which do have a predictive, analytical capability seem to focus more on computing resource consumption and performance metrics than on command and data-driven interactions between subsystems.

In this chapter we have developed an approach for modeling spacecraft subsystems and their interactions in terms of CSP process models. We treat a spacecraft as a network of subsystems which interact via commands, telemetry streams, power buses, and physical states. Our modeling approach emphasizes the construction of process models that have a structural similarity to existing ways of describing spacecraft systems, with the intent that informal concepts should have an obvious mapping into the formal domain. We have provided example-driven guidelines for modeling each of the classical spacecraft subsystems using CSP, and described the different kinds of analysis and verification that can be performed on system models built using our subsystem modeling techniques. We have demonstrated that verification run-times for a non-trivial model constructed using the approach presented in this chapter are relatively short, on the order of minutes.

Although the descriptions of subsystem modeling that appear in this chapter have focused on the classical set of spacecraft subsystems found in most spacecraft systems engineering textbooks, there is nothing in our approach that precludes a slightly different
organization of subsystems and their responsibilities. For example, some spacecraft designs combine the ADCS and Propulsion subsystems into a single Attitude and Orbit Control Subsystem (AOCS). The spacecraft system model should reflect this alternative system structure, by including an AOCS process model. Of course, the techniques used to model the individual behaviors which make up the AOCS model will be quite similar to those used for separately modeling the ADCS and Propulsion subsystems. Thus, ultimately, the modeling approach we have developed should be seen not as a rigid set of rules that prescribe a fixed methodology for modeling each subsystem, but rather as a toolkit of techniques and guidelines for modeling different aspects of subsystem behavior, and constructing spacecraft system models.
Chapter 6
Conclusions

“Writing is nature’s way of letting you know how sloppy your thinking is.”
– Richard Guindon

“Mathematics is nature’s way of letting you know how sloppy your writing is.”
– Leslie Lamport

We write requirements and design documentation because we cannot hold in our heads all of the information necessary to design a spacecraft. The act of writing the documentation often helps to clarify the requirements and the design, and to identify omissions, oversights, contradictions, and inconsistencies in our mental model of the spacecraft and its mission. Similarly, we formalize requirements and design information because we cannot concisely capture in our written documentation the level of precision necessary to allow the requirements or design to be rigorously analyzed. The act of formalization often helps to further clarify the requirements and the design, and to identify ambiguities, omissions, and inconsistencies in our informal description of the spacecraft and its mission. Moreover, the analytical tools that can be brought to bear on a formal model can be used to identify subtle errors and unforeseen interactions that might otherwise go unnoticed. Process algebra can provide spacecraft designers with a mathematical formalism for specifying, understanding, analyzing, and verifying spacecraft system behavior.

The approach to formalizing spacecraft behavior that we have developed in this dissertation involves using the process algebra CSP to construct mathematical models that describe spacecraft behavior at two different levels of abstraction: black-box specifications
of desired system behavior, and block-diagram-level models of the behavior of interacting spacecraft subsystems. Although we have emphasized the construction of models which are suitable for analysis and verification, the act of model construction alone can be valuable, since it helps to crystallize thoughts, and clarify specifications. However, as we demonstrated in the examples which appear at the end of chapters 4 and 5, bringing analytical tools to bear on a model can help to uncover additional mistakes and oversights not brought to light during the modeling process.

It must be emphasized that the specifications and models developed using the approach described in this dissertation are not in any way intended to provide a complete description of either the spacecraft requirements, or the spacecraft design. Rather, they provide another view of the system, complementary to existing views such as mass estimates, power estimates, and geometric configuration models.

6.1 Summary of Contributions

The overall contribution of the research reported in this dissertation is an exploration of the application of process algebra to the problem of specifying and verifying spacecraft system behavior, and a demonstration that such an application is both feasible and useful. A number of other, more specific contributions are also embodied in this work, including the following:

- The identification of spacecraft behavior as something that can and should be formally modeled and analyzed. There is little or no research in the literature which even considers spacecraft system behavior (as opposed to just spacecraft software behavior), and none that we have been able to find which examines the application of formal methods to such behavior.

- An example of the application of CSP to a domain to which it has not previously been applied, and a demonstration by example that CSP can be used to describe different aspects of spacecraft behavior. CSP was originally developed for modeling and analyzing concurrent software, and that has been its predominant application domain.
However, many other kinds of systems can be viewed as networks of communicating processes, and thus can potentially benefit from the application of CSP theory and tools. The work in this dissertation demonstrates that a spacecraft is one such system.

- The identification and definition of several different informally-defined specification constructs commonly used in spacecraft behavior specification, and the development of formal representations of these constructs using CSP. Not only does formalization of these constructs allow more precise specifications to be developed, but it permits all of the constructs to be understood within a common conceptual framework, and composed in meaningful ways.

- A formal, CSP-based semantics for Functional Flow Block Diagrams. FFBDs are a traditional tool of systems engineering. But the meaning of the different FFBD constructs are typically loosely defined, open to interpretation, and vary somewhat from author to author. The formal semantics for FFBDs developed in this dissertation provides a precise and unambiguous meaning for each of the standard FFBD constructs. Although the semantics presented in this dissertation is not the only possible formal semantics for FFBDs, it is the first such semantics of which we aware.

- A library of predefined CSP specification constructs suitable for developing spacecraft behavior specifications. This library provides a basis upon which to build tools which may be more accessible than raw CSP to spacecraft systems engineers.

- A taxonomy of different kinds of spacecraft subsystem interfaces, and a set of guidelines for representing each kind of interface in terms of CSP datatypes and processes. This taxonomy extends beyond the usual explicit signal interfaces to include implicit interfaces and abstract specification events.

- An approach to spacecraft system modeling which produces CSP process models that are structurally similar to informal system block diagrams. This approach makes relating spacecraft system block diagrams to the corresponding process models fairly straightforward. It permits spacecraft designers to formalize a block diagram by
assigning behavior models to each block, and using CSP parallel composition operators to specify the connections between blocks.

- A set of guidelines for modeling the behavior of spacecraft subsystems in CSP. The guidelines provide a starting point from which to develop new spacecraft system models in CSP, and a collection of techniques for developing those models. A variety of abstraction techniques for modeling key features of each of the subsystems are introduced, including abstracting the EPS as an event-dependent quantitative resource, abstracting CDH stored sequences as processes accessed via tokens, and abstracting telemetry streams as abstract state machines.

- A catalog of different kinds of behavior properties, each of which can be automatically verified in spacecraft system models developed using the aforementioned approach to subsystem modeling. Designers can select appropriate properties from this catalog, based on what it is they wish to verify about a system model. The catalog includes discussions of how to define process-based specifications for each of the properties.

- An example of the use of automated verification techniques to find errors in a spacecraft system design. This example demonstrates that the ideas and techniques developed in this dissertation can actually be applied to a practical problem, and that the resulting models do permit flaws in a spacecraft design to be discovered.

6.2 Directions for Further Research

The contributions described in the preceding section, along with the rest of the work reported in this dissertation, lay out the fundamental elements of a formal approach to specifying and verifying spacecraft behavior. However, much work can be done to refine the ideas presented herein, and to make those ideas more easily used in an industrial setting.

6.2.1 Refinements

The behavior specification approach that we developed in chapter 4 is largely based on translating existing informal concepts into a more formal structure. This approach to devel-
oping specification constructs has the advantage of providing spacecraft behavior specifiers with a familiar set of specification tools. However, experience with using the formalized specification constructs has shown that some traditional ways of specifying spacecraft behavior can be difficult to use when forced to be precise and explicit, and constraining in their lack of expressiveness when compared to directly writing specifications in CSP. This does not mean that the use of domain-specific specification constructs should be completely abandoned, but rather that it may be worthwhile to investigate other kinds of specification constructs which are better suited to precisely expressing typical elements of spacecraft behavior. Similarly, further research into the kind of spacecraft behavior properties that can and should be specified, expanding on the catalog of properties developed in chapter 5, should also be considered.

Also of interest are ways to more systematically move from a specification to a design. Some methods of formal software development provide techniques for rigorously deriving an implementation from a specification [131]. It would be interesting to explore methods for performing similar sorts of derivation to get from a system behavior specification to a system model that implements the specified behavior. A related notion is that of submodule construction [132, 133], through which the behavior for a previously undefined subsystem can be derived from a system behavior specification, and models of the other subsystems. Derivation of a complete design from a specification is most likely to be useful when developing a clean-sheet design. In contrast, submodule construction techniques are likely be helpful for determining how to adapt an existing set of subsystems to a new mission.

Another issue worthy of further investigation is the scalability of the techniques presented in this dissertation, and how that scalability can be improved. Exhaustive state-space exploration, of the kind performed by the FDR refinement checker, is prone to difficulties with state explosion, which can make analysis of a model intractable. To combat this problem, we have put much effort into devising subsystem modeling approaches that capture the essential behavior of each subsystem without producing an unnecessary proliferation in the number of states. However, case studies on real spacecraft designs are necessary to really
know how effective that effort has been. Depending on the outcome of the case studies, it may, as we mentioned in chapter 5, become necessary to incorporate hierarchical state-space compression techniques [129] as an integral part of the CSP-based approach to spacecraft behavior modeling. Since the effectiveness of hierarchical compression depends on the order in which the different elements of a process network is composed [3], it would be useful to test different ways of assembling system behavior specifications and system models to see which give the best runtime for refinement checks. Other existing techniques in the CSP literature for mitigating state-space explosion should also be evaluated, to determine whether they are applicable to the kinds of models we use to represent spacecraft behavior.

In addition to developing refined methods of applying the CSP-based approach described in this dissertation, another potential direction for research is the translation of the ideas introduced here into other formalisms. For example, the industrially popular LOTOS specification language shares much in common with CSP [134], but is associated with an alternative set of analysis and verification tools to those available for CSP [135]. Efforts to model highly-reconfigurable spacecraft might benefit from the dynamic process topologies provided by the π-calculus [62], although the lack of multi-way synchronization in the π-calculus is likely to require the use of a modeling approach with substantial differences to the one developed in this dissertation. Extension of the present work to hybrid CSP, or some other hybrid process algebra [136], would allow a closer integration of the discrete and continuous aspects of spacecraft behavior.

6.2.2 Practicalities

Several things can be done to make the techniques we have presented in this dissertation easier to apply in an industrial setting. One of these things is an expansion and systematization of the guidelines we have developed for modeling spacecraft subsystems. Given a greater amount of experience with attempting to model different kinds of spacecraft, it should become possible to develop a pattern language [137] of subsystem modeling techniques. Such a pattern language would make the task of subsystem modeling easier, by providing not just a set of modeling techniques, but also guidance on when to apply a
particular technique, and indications of which other techniques might also be helpful. Ultimately, a pattern-based approach to spacecraft subsystem behavior design may benefit not just the model development process, but also the spacecraft design process itself, by helping designers to sift through different behavior options available to them, and find those most suitable for the design context in which they are working.

Another worthwhile line of research is the development of one or more domain-specific languages (DSLs) for describing spacecraft behavior, and tools for translating these languages into corresponding CSP models constructed using the methods developed in this dissertation. These DSLs may be text-based, graphical, or some combination of the two. Providing a DSL for spacecraft behavior makes it possible to shield end-users of the language from the many internal details of our modeling approach which are necessary to produce a useful CSP model, but which are irrelevant to the task of spacecraft behavior definition. As a result, spacecraft behavior models should become easier to produce, and much more readable. A good example of the use of a DSL to make the model development task easier is the Casper toolkit [47], which facilitates the modeling of security protocols using a language similar to standard protocol description techniques, and automatically translates the protocol descriptions into a set of CSP process models. Eames et al. [80] describe a graphical DSL and associated tools which permit model-building using a subset of the ideas presented in chapter 4 of this dissertation.

Research into improving the integration of our CSP-based approach to spacecraft behavior modeling into the rest of the spacecraft design process is also likely to be beneficial. A number of opportunities exist for greater integration of CSP into the spacecraft design process. For example, the literature already includes work that facilitates the verification of digital designs developed in languages such as VHDL and occam against CSP specifications [83,85,138], as well as methods for deriving software and hardware designs from CSP specifications [44,68,124]. However, further investigation into how to relate our CSP models to those required as input to the existing CSP tools and methods for hardware and software design is necessary. Similarly, there exist tools for automatically testing hardware and soft-
ware in accordance with specifications written in CSP [36, 139], but it is not immediately clear how those specifications relate to the models produced using our approach. Finding a way to translate parts of our CSP-based models into MATLAB’s Stateflow® [140] would permit the combined simulation of both the discrete and continuous dynamics of various hybrid spacecraft elements, such as attitude controllers, while ensuring that the discrete portions of the model accurately reflected the system-level assumptions made about them. Finally, efforts to translate existing work on formalizing and model-checking spacecraft autonomy systems [32, 90, 141] into a CSP framework would enable designers to rigorously verify that the behaviors of the spacecraft autonomy system and the rest of the spacecraft system are compatible.

### 6.2.3 Other Directions

Although we have focused on spacecraft systems engineering in this dissertation, many of the techniques developed herein can be applied to a much wider range of systems engineering problems. Possible directions for future research in that area include:

- Developing a formal, CSP-based semantics for the behavior diagrams found in the systems engineering modeling language SysML [142]. SysML includes a FFBDs as one of its modeling views, although those FFBDs appear to have a slightly different semantics than we have presented here. It would be interesting to be able to compare both sets of FFBD semantics through formal representations of each.

- Investigating the relationship between the CSP-based approach to system modeling and Wymore’s hierarchical state-machine-based approach [6]. Wymore’s approach also includes considerations such as technology requirements, test requirements, and rigorous design trade-offs, all of which are well outside the scope of the work presented in this dissertation, but of which it would be useful to take advantage.

- Generalizing the spacecraft-specific elements of our approach into something more suitable for modeling of arbitrary systems. Alternatively, it may be more practical to develop guidelines for generating domain-specific modeling approaches.
6.3 Envisioned Place Within the Design Process

Formal modeling of spacecraft behavior using the techniques described in this dissertation is likely to provide the most benefit during the conceptual and preliminary design phases of a spacecraft development program, when the essential system-level behavior of the spacecraft is defined and refined. By developing formal specifications of the intended system behavior, spacecraft designers can flush out problems and ambiguities in the behavior before those problems have a significant impact on the evolving spacecraft design. Similarly, by constructing and verifying models of the behavior of a proposed system design, spacecraft designers can gain greater confidence that their basic concept for the way in which the different subsystems will interact is sound, before investing significant time and effort in the detailed design of each subsystem.

The techniques we have laid out for modeling each of the subsystems largely treat those subsystems as black-boxes, and ignore the details of how the subsystems internally operate. However, the resulting subsystem models can themselves be used as specification processes, against which more detailed, design-oriented subsystem models can be verified. For example, we might decompose a CDH subsystem into processes representing the different circuit boards which make up the design, and verify the composite model of the CDH subsystem against our black-box CDH process model. As long as the detailed CDH model refines the abstract, black-box model of the CDH subsystem we can, due to the transitivity and monotonicity of refinement relations, be confident that the detailed design will interact correctly with the rest of the spacecraft, without needing to perform a direct verification of those interactions. Compositional reasoning of this sort can be used to make tractable the verification of quite complex systems.

An end-to-end application of the approach developed in this dissertation might involve:

- Development of a formal system behavior specification, based on mission scenarios, operational concepts, and informally expressed requirements. At the same time as the specification is developed, formal expressions of mission scenarios, and of desirable system behavior properties may also be developed.
• Construction of process models defining the intended behavior of each subsystem, and construction from those subsystem models of a spacecraft system model representing the proposed spacecraft design.

• Verification that the system behavior specification is consistent, and provides the desired operational behavior, and that the spacecraft system model possesses the desired behavior properties, and implements the system behavior specification. Achieving all of these goals will likely involve iterative refinement of both the specification and the system design.

• Use of the subsystem process models as a part of the specification from which the detailed subsystem designs are developed. Continued refinement of the specification and system design as the subsystem designs are fleshed out, and necessary changes in subsystem interactions are identified.

Of course, it is not necessary to perform all of the preceding tasks in every spacecraft development project. A completely formal development of the kind just described may be considered too burdensome for some projects. In those cases, some benefit may still be derived by applying only parts of the approach. For example, some design teams may choose to construct a system model, but not bother with developing a complete formal system behavior specification, and instead verify the system model against a suite of individual properties and scenarios. The approach described in this dissertation provides a framework and method for thinking clearly about the potentially confusing and ambiguity-prone problems of defining spacecraft system behavior, and a toolkit from which spacecraft designers can select those tools which they feel will provide the most benefit to the development of a spacecraft behavior design.
References


[82] A. E. Abdallah and I. W. Damaj, “Reconfigurable hardware synthesis of the
IDEA cryptographic algorithm,” in Communicating Process Architectures 2004,

an occam-to-FPGA compiler,” in Communicating Process Architectures 2004,

uk/pub/amulet/theses/bardsley_phd.pdf.

[85] X. Wang, M. Kwiatkowska, G. Theodoropoulos, and Q. Zhang, “Towards a
unifying CSP approach for hierarchical verification of asynchronous hardware,”
in Proceedings of the Fourth International Workshop on Automated Verification

Communicating Process Architectures 2000, P. P. H. Welch and A. W. P. Bakkers,

[87] G. Lowe, “Breaking and fixing the Needham-Schroeder public-key protocol using
FDR,” in Tools and Algorithms for the Construction and Analysis of Systems

checking,” in Proceedings of ESORICS’02, ser. Lecture Notes in Computer Science,


integrated formal method for intelligent swarms,” in Proceedings of the 10th Interna-

[91] G. J. Holzmann, “The model checker SPIN,” IEEE Transactions on Software Engi-


data handling flight software requirements specification, draft,” NASA Goddard

[120] C. A. Grasso, “The fully programmable spacecraft: procedural sequencing for JPL
deep space missions using VML (Virtual Machine Language),” in Proceedings of the

[121] E. Gat, “Non-linear sequencing,” in Proceedings of the 1999 IEEE Aerospace Confer-
dence, 1999.

paal - present and future,” in Proceedings of 40th IEEE Conference on Decision and

[123] A. I. McInnes, “Design and implementation of a proof-of-concept MMORPG using
CSP and occam-π,” in Proceedings of the 2005 International Conference on Parallel
and Distributed Processing Techniques and Applications (PDPTA’05), H. Arabnia,

[124] A. A. McEwan, “A calculated implementation of a control system,” in Communicating


[127] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, “Basic concepts and tax-
onomy of dependable and secure computing,” IEEE Transactions on Dependable and

tributed systems,” QinetiQ, FORWARD Deliverable D9, June 2004.

[129] A. Roscoe, M. Goldsmith, P. Gardiner, J. Hulance, D. Jackson, and J. Scatter-
good, “Hierarchical compression for model-checking CSP or how to check 10^20 dining

dog transformations for property-oriented model checking,” in Proceedings of Formal


[132] L. Lai and J. W. Sanders, “A weakest-environment calculus for communicating pro-


Appendices
Appendix A

Spacecraft Behavior Framework Library

{-

---- Spacecraft Behavior Framework ------------------------------------

A library of building blocks for constructing spacecraft behavior
specifications. This library can be 'included' into other CSP scripts.
The process definitions contained herein can then be used to build
spacecraft behavior specifications.

CSPm seems to have a nasty habit of capturing non-locally defined
names inside process definitions, even if a local version of those
names is provided as a process parameter. To avoid this problem, every
process parameter has been tagged with a "namespace" prefix which
should reduce the likelihood of name conflicts. This is ugly, but
seems to work. The "namespace" prefix is "SCBF_

-}

{-

---- Lifting functions ------------------------------------------------

__LiftF__ 'lifts' a function 'f' to a process which receives values
from the input set of 'f' over channel 'in', and sends corresponding
values from the output set of 'f' over channel 'out'. The function 'f'
is defined using CSPm's embedded functional programming language.

-}
The \_\_LiftF2\_\_ process provides the same behavior as the 'LiftF' processes, but allows the function 'f' to be provided in terms of a binary relation (i.e. a set of tuples) rather than directly as a function definition.

\_\_MIMOLiftF\_\_ behaves similarly to LiftF, but is intended for use with MI or MO functions, which must have some kind of input sequencing strategy associated with them. The 'reqin' channel is used to request that an MI behavior obtain a group of inputs. The 'ackout' channel is used to receive an acknowledgment from an MO behavior that a group of outputs has been completely transmitted.

\_\_SwitchedLiftF\_\_ is a modal lifted function that provides the behavior of LiftF when in the 'on' mode, and acts as a sink for input values when in the 'off' mode. The mode is changed by sending an 'on' or 'off' symbol through the 'switch' channel. The initial mode is 'off'.

LiftF(SCBF\_in, SCBF\_out, f) =
SCBF\_in?x -> SCBF\_out!f(x) -> LiftF(SCBF\_in, SCBF\_out, f)

LiftF2(SCBF\_in, SCBF\_out, f) =
SCBF\_in?x -> SCBF\_out!apply(f,x) -> LiftF2(SCBF\_in, SCBF\_out, f)

MIMOLiftF(SCBF\_in, SCBF\_out, SCBF\_reqin, SCBF\_ackout, f) =
let
SCBF_MLF = SCBF_reqin -> SCBF_in?x -> SCBF_out!f(x) 
    -> SCBF_ackout -> SCBF_MLF

within

SCBF_MLF

SwitchedLiftF(SCBF_in, SCBF_out, SCBF_f, SCBF_switch) =

let

    SCBF_SLF(s) =
        s == off & SCBF_in?_ -> SCBF_SLF(s) 
        []
        s == on & SCBF_in?x -> SCBF_out!SCBF_f(x) -> SCBF_SLF(s) 
        []
        SCBF_switch?s' -> SCBF_SLF(s')

within

    SCBF_SLF(off)

{-

    ---- MI/MO behaviors --------------------------------------

    Generic 2I/2O behavior specifications.

    __SeqIn2__ awaits a request in the 'reqin' channel, following which
    it receives values from channel 'in1' and then 'in2'. The two values
    are aggregated into a 2-tuple which is output through channel 'tuple'.

    __SeqOut2__ receives a 2-tuple through channel 'tuple'. It sends the
    first element of the tuple through 'out1', then the second element
through 'out2', then sends an 'ackout' signal.

__ParIn2__ awaits a request in the 'reqin' channel, following which it receives values from channels 'in1' and 'in2'. The two values are aggregated into a 2-tuple which is output through channel 'tuple'.__

__ParOut2__ receives a 2-tuple through channel 'tuple'. It sends the first element of the tuple through 'out1', the second through 'out2', then sends an 'ackout' signal when both outputs have completed.

```
SeqIn2(SCBF_reqin, SCBF_in1, SCBF_in2, SCBF_tuple) =
    SCBF_reqin -> SCBF_in1?x -> SCBF_in2?y -> SCBF_tuple!(x,y)
    -> SeqIn2(SCBF_reqin, SCBF_in1, SCBF_in2, SCBF_tuple)

SeqOut2(SCBF_tuple, SCBF_out1, SCBF_out2, SCBF_ackout) =
    SCBF_tuple?(x,y) -> SCBF_out1!x -> SCBF_out2!y -> SCBF_ackout
    -> SeqOut2(SCBF_tuple, SCBF_out1, SCBF_out2, SCBF_ackout)

ParIn2(SCBF_reqin, SCBF_in1, SCBF_in2, SCBF_tuple) =
    let
        X = extensions(SCBF_in1)
        Y = extensions(SCBF_in2)
        In1 = SCBF_reqin -> SCBF_in1?x -> [] y:Y @ SCBF_tuple!(x,y) -> In1
        In2 = SCBF_reqin -> SCBF_in2?y -> [] x:X @ SCBF_tuple!(x,y) -> In2
    within
        In1 [! {SCBF_reqin, SCBF_tuple}] |] In2
```
ParOut2(SCBF_tuple, SCBF_out1, SCBF_out2, SCBF_ackout) =

let

Out1 = SCBF_tuple?(x,y) -> SCBF_out1!x -> SCBF_ackout -> Out1
Out2 = SCBF_tuple?(x,y) -> SCBF_out2!y -> SCBF_ackout -> Out2

within

Out1 [ | { | SCBF_ackout, SCBF_tuple | } | ] Out2

{-

---- Event Sequence -----------------------------------------------

__EventSeq__ specifies a sequential execution of events in the list of events 'S'.

-}

EventSeq(SCBF_S) = ; e:SCBF_S @ e -> SKIP

{-

---- State Transition System ---------------------------------------

__StateTransitions__ defines a state transition system in terms of transitions between different states, and the events that can trigger those transitions. The initial state is 's0'. The auxiliary channel 'transition' is used for signaling state transitions to other processes. 'TransDefs' is a set of 3-tuples defining state transitions in terms of current state, transition trigger event, and new state.

-}
StateTransitions(SCBF_s0, SCBF_transition, SCBF_TransDefs) =
    let
        -- Find the transitions definitions for state 's'
        Trans(s) = { (e,s') | (state,e,s') <- SCBF_TransDefs, s == state }
        -- The transition behavior for state 's'
        State(s) =
            [] (e,s'):Trans(s) @ e -> SCBF_transition.s' -> State(s')
    within
        SCBF_transition.SCBF_s0 -> State(SCBF_s0)

{-
    ---- Event-Triggered Behavior ------------------------------------------

    EventTrigger(SCBF_Triggers, SCBF_P) = [] t:SCBF_Triggers @ t -> SCBF_P

    ---- Assignable State ------------------------------------------------

    AssignableState is a process encapsulating a state value ('val')
    that may be either set through the channel 'set', or read through the
channel 'get'. Assignment of a new state value generates a signal on the channel 'trans'. The initial value of 'val' is 'init'.

```ml
AssignabeState(SCBF_set, SCBF_get, SCBF_trans, SCBF_init) =
  let
    State(val) =
      (SCBF_get!val -> State(val))
      []
    (SCBF_set?val'
      -> if (val' != val)
        then (SCBF_trans!val' -> State(val'))
        else State(val))
  within
    State(SCBF_init)
```

{-

---- Quantitative Resource ---------------------------------------------

__QuantResource__ is a process encapsulating an integer quantitative value ('val'). This value has upper and lower bounds 'max' and 'min'. The value can be changed by sending an integer-valued magnitude of change through channel 'delta', and read through channel 'get'. Changes in the value result in a signal on channel 'trans'. The initial value of 'val' is 'init'. Changes in the value that result in 'val' exceeding the upper or lower bounds result in a corresponding signal on channel 'qr_exception' and termination of
the process.

datatype ResourceException = resource_overflow | resource_underflow

channel qr_exception : ResourceException

QuantResource(SCBF_delta, SCBF_get, SCBF_trans, SCBF_min, SCBF_max, SCBF_init) =

let
    Range = {SCBF_min..SCBF_max}
    Quantity(val) =
        val > SCBF_max & qr_exception.resource_overflow -> STOP
        []
        val < SCBF_min & qr_exception.resource_underflow -> STOP
        []
        member(val, Range) & (SCBF_get!val -> Quantity(val))
        []
        member(val, Range) &
        (SCBF_delta?d ->
            let
                val' = val + d
            within
                if (val' != val) and member(val', Range)
                then (SCBF_trans!val' -> Quantity(val'))
                else Quantity(val'))
        within
            Quantity(SCBF_init)
{-

---- Bounded Blocking Buffer -----------------------------------------

__BoundedBlockingBuffer__ is a standard CSP buffer process. It accepts up to 'N' values on channel 'in', after which it refuses further inputs until an output has occurred. Values are output on channel 'out'. Output events are refused when the buffer is empty.

-}

BoundedBlockingBuffer(SCBF_in, SCBF_out, SCBF_N) =

let

  Buff(s) =

  (#s < SCBF_N & SCBF_in?x -> Buff(s^<x>))

  []

  (#s > 0 & SCBF_out!head(s) -> Buff(tail(s)))

within

  Buff(<>)

{-

---- Generic constraints ---------------------------------------------

We can define two quite general constraint processes which are useful for capturing a variety of different sequencing requirements. These constraint processes are 'Between', and 'Outside'. The 'Between' and 'Outside' constraint processes allow events in the set 'Ev' to occur only in the interval between the occurrence of some enabling event and
the next occurrence of a disabling event. The difference between the
two constraints is that 'Between' assumes that events in 'Ev' are
initially disabled, while 'Outside' assumes that they are initially
enabled. These constraint processes can be defined in terms of each
other. The design of these processes is due to Roscoe.

{-

Between(SCBF_En, SCBF_Dis, SCBF_Ev) =
     ([] x:SCBF_En @ x -> Outside(SCBF_Dis, SCBF_En, SCBF_Ev))
     []
     ([] x:SCBF_Dis @ x -> Between(SCBF_En, SCBF_Dis, SCBF_Ev))

Outside(SCBF_Dis, SCBF_En, SCBF_Ev) =
     ([] e:SCBF_Ev @ e -> Outside(SCBF_Dis, SCBF_En, SCBF_Ev))
     []
     ([] x:SCBF_En @ x -> Outside(SCBF_Dis, SCBF_En, SCBF_Ev))
     []
     ([] x:SCBF_Dis @ x -> Between(SCBF_En, SCBF_Dis, SCBF_Ev))

{-

    ---- Mode Constraint ----------------------------------

__ModeConstraint__ is a 'Between' constraint that enables events in
'InitEv' between any mode transition 'modetrans' from the 'Enable'
set, and the next transition to a mode in the 'Disable' set.

-}
ModeConstraint(SCBF_InitEv, SCBF_modetrans, SCBF_Enable, SCBF_Disable) =

let

    SCBF_En = { SCBF_modetrans.m | m <- SCBF_Enable }
    SCBF_Dis = { SCBF_modetrans.m | m <- SCBF_Disable }

within Between(SCBF_En, SCBF_Dis, SCBF_InitEv)

{-

    ---- FFBD building blocks ------------------------------------------------------

Building block process for translating FFBDs into CSP constraint processes. Each process corresponds to a standard FFBD graphical element.


-}

-- A single function block, characterized by the input and output
-- channels of the represented function
FFBDblock(SCBF_in, SCBF_out) =

    [] i:{|SCBF_in|} @ i -> ([] o:{|SCBF_out|} @ o -> SKIP)

-- FFBD sequencing (i.e. the arrows between blocks)
FFBDseq(SCBF_FFBD_1, SCBF_FFBD_2) = SCBF_FFBD_1; SCBF_FFBD_2

-- FFBD concurrency - the AND bubble
FFBDand(SCBF_FFBDset) = ||| FFBD:SCBF_FFBDset @ FFBD
-- FFBD selection - the OR bubble

\[ \text{FFBDor}(\text{SCBF\_FFBDset}) = \emptyset \text{ FFBD:SCBF\_FFBDset} \] 

\[ \text{FFBD iteration} \]

\[ \text{FFBDiteration}(\text{SCBF\_test\_state}, \text{SCBF\_GoSet}, \text{SCBF\_FFBD}) = \]

\[
\begin{align*}
\text{let} \\
\text{Loop} = \\
\text{FFBDseq}(\text{SCBF\_FFBD}, \\
\text{FFBDchoice}(\text{SCBF\_test\_state}, \text{SCBF\_GoSet}, \text{SKIP}, \text{Loop})) \\
\text{within} \\
\text{Loop}
\end{align*}
\]

{-

--- Temporal constraints

Temporal constraints provide a way to define timing relationships between different system events.

-}

-- Timeline constructor
Timeline(SCBF_T) = ; t:SCBF_T @ t -> SKIP

-- Before constraint - 'event' _must_ occur before 'time', and only once
MustBeforeTime(SCBF_event, SCBF_time) = SCBF_event -> SCBF_time -> SKIP

-- Events in 'Ev' may occur before 'time', but not after
MayBeforeTime(SCBF_Ev, SCBF_time) = Outside({SCBF_time}, {}, SCBF_Ev)

-- Events in 'Ev' may occur after 'time', but not before
MayAfterTime(SCBF_Ev, SCBF_time) = Between({SCBF_time}, {}, SCBF_Ev)

-- Events in 'Ev' may occur between 'time1' and 'time2'
MayBetweenTimes(SCBF_Ev, SCBF_time_1, SCBF_time_2) =
   Between({SCBF_time_1}, {SCBF_time_2}, SCBF_Ev)

{-
   ---- Constraint network --------------------------------------------

   __ConstraintNet__ is a parallel composition of constraints. 'Constr'
   is a set of constraints, each defined as a 2-tuple consisting of
   an alphabet set and a constraint process.

   __aConstraintNet__ is the alphabet of the corresponding
   'ConstraintNet' process.
-

aConstraintNet(SCBF_Constr) = Union({ aC | (aC,C) <- SCBF_Constr })
ConstraintNet(SCBF_Constr) = (|| (aC,C):SCBF_Constr @ [aC] C)

{-
  ---- Generic subsystem datatypes ----------------------------------------

  The __OnOff__ and __StateIF__ datatypes are convenience datatypes
  for use in subsystem model construction.
-}

datatype OnOff = on | off
datatype StateIF = setval | getval | trans

{-
  ---- State Telemetry Stream ---------------------------------------------

  The __State Telemetry Stream__ process is constructed under the
  assumption that the data contained in the telemetry stream
  represents some kind of subsystem state information. Transitions
  ('state_trans') in the subsystem state produce corresponding
  transitions ('stream_trans') in the telemetry data state. The
  telemetry stream itself may be either inactive, in which case
  telemetry data is unavailable, or active, in which case telemetry
  data is available to consumers of the telemetry stream. Signals on a
  control channel ('mode') allow the telemetry stream
  to be switched to and from its inactive state.
-}

datatype GenericStreamState = stream_inactive
StateTelemetryStream(SCBF_mode, SCBF_state_get, SCBF_state_trans,
SCBF_stream_set, SCBF_stream_get, SCBF_stream_trans, SCBF_f) =

let
StreamState =
    AssignableState(SCBF_stream_set, SCBF_stream_get,
SCBF_stream_trans, stream_inactive)

Inactive =
    SCBF_state_trans?_ -> Inactive
    []
    SCBF_mode.off -> Inactive
    []
    SCBF_mode.on -> Setup

Setup =
    SCBF_state_trans?x -> SCBF_stream_set!SCBF_f(x) -> Active
    []
    SCBF_state_get?x -> SCBF_stream_set!SCBF_f(x) -> Active
    []
    SCBF_mode.off -> SCBF_stream_set.stream_inactive -> Inactive
    []
    SCBF_mode.on -> Setup

Active =
    SCBF_state_trans?x -> SCBF_stream_set!SCBF_f(x) -> Active
    []
SCBF_mode.off -> SCBF_stream_set.stream_inactive -> Inactive
[]
SCBF_mode.on -> Active
within
Inactive
[SCBF_stream_set <-> SCBF_stream_set]
StreamState

{-

------ Subsystem Mode Power -----------------------------------------

The _Subsystem Mode Power_ process translates transitions in the
operating mode of a subsystem ('modetrans') into transitions in the
quantity of power consumed by the subsystem ('power_delta').
-

datatype ModeTransDelimiter = begin | end

SubsysModePower(SCBF_initmode, SCBF_modetrans,
                   SCBF_power_delta, SCBF_f_ModePower) =
let
  f_PowerDelta(m,m') = SCBF_f_ModePower(m') - SCBF_f_ModePower(m)

  PMode(m) =
  SCBF_modetrans.begin?m'
  -> if m != m'
    then (SCBF_power_delta!f_PowerDelta(m,m')
Auxiliary functions for working with spacecraft functions defined as binary relations. These are mostly inspired by the Z specification language.

-- Find the domain of relation 'rel'
\[
\text{dom}(\text{SCBF\_rel}) = \{ d \mid (d,r) \in \text{SCBF\_rel} \}
\]

-- Find the range of relation 'rel'
\[
\text{range}(\text{SCBF\_rel}) = \{ r \mid (d,r) \in \text{SCBF\_rel} \}
\]

-- The relation that results when we restrict the domain of 'rel' to 'dset'
\[
\text{dom\_restrict}(\text{SCBF\_dset}, \text{SCBF\_rel}) =
\{ (d,r) \mid (d,r) \in \text{SCBF\_rel}, \text{member}(\text{SCBF\_dset}, d) \}
\]

-- The relation that results when we restrict the range of 'rel' to 'rset'
\[
\text{range\_restrict}(\text{SCBF\_rset}, \text{SCBF\_rel}) =
\{ (d,r) \mid (d,r) \in \text{SCBF\_rel}, \text{member}(\text{SCBF\_rset}, r) \}
\]

-- Apply the relation 'func' to the value 'val'
-- We assume here that 'func' is a relation that is a partial function
apply(SCBF_func, SCBF_val) =

   let
      pick({x}) = x

   within pick({ r | (d,r) <- SCBF_func, d == SCBF_val })
Appendix B

Machine-Readable CSP Translation of Example 4.3.3.1

include "sc-behavior-framework.csp"

-- Type and channel defs
datatype Attitude = tumbling | earth_limb_scan | sun_pointing
datatype AttitudeCommand = detumble | science_attitude | safe_attitude
datatype Mode = launch | safe | nominal
datatype Command = att.AttitudeCommand | mode.Mode
datatype Measurement = sample
nametype ScienceData = (Measurement,Attitude)
datatype Data = formatted.ScienceData
datatype DownlinkMsg = modestatus.Mode | attstatus.Attitude
| missiondata.Data | fault_occurred
datatype Deployables = antenna.{1..2} | solar_array

-- External channels
channel cmd : Command
channel instrument : Measurement
channel downlink : DownlinkMsg
channel separation

-- Internal channels
channel attitude, sense_attitude, attitude_transition: Attitude
channel in_Sci : ScienceData
channel sci_req, sci_out_ack
channel trans_Mode : Mode
channel deploy : Deployables
channel fault

SCspec =

let

-- Separation behavior
Deployments = EventSeq<deploy.solar_array, deploy.antenna.1,
                     deploy.antenna.2>

SeparationBehavior =
    EventTrigger({separation}, Deployments)

-- Spacecraft functions
f_Att(c) =
    let
        f = {(detumble, sun_pointing),
             (science_attitude, earth_limb_scan),
             (safe_attitude, sun_pointing)}
    within apply(f,c)

f_Sci((m,a)) = formatted.(m,a)

-- Lifted spacecraft functions
AttitudeCommanding = LiftF(cmd.att, attitude, f_Att)

Science' = (SeqIn2(sci_req, instrument, sense_attitude, in_Sci)
MIMOLiftF(in_Sci, downlink.missiondata,
sci_req, sci_out_ack, f_Sci))
\{|in_Sci, sci_req, sci_out_ack|}

-- Fault response
FaultResponse =

   EventTrigger({fault},
   EventSeq(<attitude.sun_pointing,
     downlink.fault_occurred>));

   FaultResponse

-- Spacecraft mode transition behavior
SC_Modes =

   let
     TransitionDefs = {{launch, separation, safe},
                      (safe, cmd.mode.nominal, nominal),
                      (nominal, cmd.mode.safe, safe),
                      (nominal, fault, safe)}
     within
       StateTransitions(launch, trans_Mode, TransitionDefs)

   f_ReportMode(m) = modestatus.m

   SC_Modes' = (SC_Modes
     [[|{trans_Mode}|]]
     LiftF(trans_Mode, downlink, f_ReportMode))

-- States
AttitudeState =
   AssignableState(attitude, sense_attitude,
   attitude_transition, tumbling)

-- Mode constraints
aAtt_Modes = {{cmd.att, trans_Mode}}

Att_Modes =
   ModeConstraint({{cmd.att}}, trans_Mode,
   {safe, nominal}, {launch})

aScience_Modes = {{instrument, trans_Mode}}

Science_Modes =
   ModeConstraint({{instrument}}, trans_Mode,
   {nominal}, {launch, safe})

-- Other constraints
aNoAttChange = {{instrument, downlink.missiondata, attitude}}

NoAttChangeDuringScience =
   Outside({{instrument}}, {{downlink.missiondata}},
   {{attitude}})

aNoDL = {{downlink, attitude_transition}}

NoDLWhileTumbling =
   Between({attitude_transition.sun_pointing,
   attitude_transition.earth_limb_scan},
   {attitude_transition.tumbling}, {{downlink}})

aNoDownlinkBeforeSep = {{downlink, separation}}
NoDownlinkBeforeSep = Between({separation}, {}, {downlink})

-- Constraint network
ConstraintSet = {(aAtt_Modes, Att_Modes),
(aScience_Modes, Science_Modes),
(aNoAttChange, NoAttChangeDuringScience),
(aNoDL, NoDLWhileTumbling),
(aNoDownlinkBeforeSep, NoDownlinkBeforeSep)}
aCNet = aConstraintNet(ConstraintSet)
CNet = ConstraintNet(ConstraintSet)

within
((((((AttitudeCommanding -- spacecraft behaviors
    ||| Science')
    ||| SC_Modes')
    [{separation}]
    SeparationBehavior)
    [{fault}]
    FaultResponse))
    [{attitude, sense_attitude}] -- behavior/state interface
AttitudeState)
    [{aCNet}]
CNet) -- the constraint network
Appendix C

Machine-Readable CSP for the Example Specification from “A Model-Based Design Tool for Systems-Level Spacecraft Design”

The following machine-readable CSP was generated for the paper “A Model-Based Design Tool for Systems-Level Spacecraft Design” [80], which was presented at the 2006 AIAA/USU Conference on Small Satellites. Among other things, the specification demonstrates the integration of mode-transition systems and FFBDs. There are some idiosyncrasies in the datatypes (notably the lack of compound data values) which were necessitated by limitations in the version of the prototype graphical modeling tool which the paper described.

include "sc-behavior-framework.csp"

-- Types
datatype Attitude = sun_pointing | target_attitude_1
    | target_attitude_2 | earth_pointing
datatype AttitudeCommand = safehold | science_target_1
    | science_target_2 | science_standby
datatype Mode = standby | science | wait_to_science | wait_to_standby
datatype Command = att.AttitudeCommand | mode.Mode
datatype Measurement = datum
nametype ScienceData = (Measurement,Attitude)

-- External channels
channel cmd_att : AttitudeCommand
channel cmd_mode : Mode
channel instrument : Measurement
channel data : ScienceData

-- Internal channels
channel set_attitude, sense_attitude, trans_attitude : Attitude
channel trans_mode : Mode
channel get_modestate, trans_modestate : Mode
channel begin_ffbd, end_ffbd : {standby, science}
channel in_Sci : ScienceData
channel sci_in_req, sci_out_ack

SCspec =
  let
    -- Spacecraft functions
    -- The attitude mapping
    f_Att(c) =
      let
        f = {(safehold, sun_pointing),
            (science_target_1, target_attitude_1),
            (science_target_2, target_attitude_2),
            (science_standby, earth_pointing)}
      within apply(f,c)

    -- The science mapping
    f_Sci(x) = x  -- identity function
-- Lifted spacecraft functions

\[
\text{AttitudeCommanding} = \text{LiftF}(\text{cmd\_att, set\_attitude, f\_Att})
\]

-- SeqIn2 is the "input strategy" that implements the input merging
-- in_Sci is the internal channel that passes the tuple from the
-- input strategy to the actual lifted function

\[
\text{Science} = (\text{SeqIn2}(\text{sci\_in\_req, instrument, sense\_attitude, in\_Sci})
\]
\[
[|\{|\text{sci\_in\_req, in\_Sci}\}|]
\]
\[
\text{MIMOLiftF}(\text{in\_Sci, data, sci\_in\_req, sci\_out\_ack, f\_Sci})
\]
\[
\{|\text{in\_Sci}|| -- hide the internal channel
\]

-- State transition system defining spacecraft mode transition
-- behavior. Note the "wait" states necessary to enforce
-- exclusive execution of the FFBDS

\[
\text{InitialMode} = \text{standby}
\]

\[
\text{SC\_Modes} =
\]
\[
\text{let}
\]
\[
\text{TransitionDefs} =
\]
\[
\{(\text{standby, cmd\_mode\_science, wait\_to\_science}),
\]
\[
(\text{wait\_to\_science, end\_ffbd\_standby, science}),
\]
\[
(\text{science, cmd\_mode\_standby, wait\_to\_standby}),
\]
\[
(\text{wait\_to\_standby, end\_ffbd\_science, standby})
\]
\[
\text{within}
\]
\[
\text{StateTransitions(InitialMode,trans\_mode,TransitionDefs)}
\]

-- Shared state objects
AttitudeState =
    AssignableState(set_attitude, sense_attitude, trans_attitude, sun_pointing)

ModeState =
    AssignableState(trans_mode, get_modestate, trans_modestate, InitialMode)

-- Standby Mode FFBD
StandbyFFBD =
    let
      AttitudeBlock = FFBDblock(cmd_att, set_attitude)
      aFFBD = {|cmd_att, set_attitude|}

      -- Note that diff(Modes,{standby}) is a set difference
      -- operation, so the "GoSet" (i.e. loop termination
      -- condition) is any mode *except* standby

      -- The FFBD is can *always* signal "end" if it’s not
      -- active, but must terminate before it can signal
      -- "end" if it *is* active - we need to do this to
      -- prevent commanded mode transitions from blocking in
      -- the wait state in cases where the FFBD process
      -- hasn’t become active before the command arrives

      FFBD =
          (end_ffbd.standby -> FFBD)
          []
          (begin_ffbd.standby ->
FFBDiteration(get_modestate, diff(Mode,{standby}),
   AttitudeBlock);
end_ffbd.standby -> FFBD)

within
   AttitudeCommanding
   [[aFFBD]]
   FFBD

-- Science Mode FFBD
ScienceFFBD =
   let
      AttitudeBlock = FFBDblock(cmd_att, set_attitude)
      ScienceBlock = FFBDblock(sci_in_req, sci_out_ack)
      aFFBD = {cmd_att, set_attitude, sci_in_req, sci_out_ack}
      FFBD =
         (end_ffbd.science -> FFBD)
         []
         (begin_ffbd.science ->
            FFBDiteration(get_modestate, diff(Mode,{science}),
               FFBDor({AttitudeBlock, ScienceBlock}));
            end_ffbd.science -> FFBD)
   within
      ((AttitudeCommanding ||| Science)
       [[aFFBD]]
       FFBD) \ {sci_in_req, sci_out_ack}

-- Mode constraints
-- This is the "alphabet" for the mode constraint, i.e. all events
-- that it must synchronize on.
aStandbyFFBDConstraint = {{begin_ffbd.standby,trans_mode}}

-- This mode constraint says that the event begin_ffbd.standby is
-- enabled between any occurrence of the event trans_mode.standby,
-- and the next occurrence of any other event on trans_mode
StandbyFFBDConstraint = 
    ModeConstraint({begin_ffbd.standby},
                   trans_mode, {standby}, diff(Mode, {standby}))

aScienceFFBDConstraint = {{begin_ffbd.science,trans_mode}}
ScienceFFBDConstraint = 
    ModeConstraint({begin_ffbd.science},
                   trans_mode, {science}, diff(Mode, {science}))

-- Constraint network
ConstraintSet = {(aStandbyFFBDConstraint, StandbyFFBDConstraint),
                   (aScienceFFBDConstraint, ScienceFFBDConstraint)}
aCNet = aConstraintNet(ConstraintSet)
CNet = ConstraintNet(ConstraintSet)

within 
  (((SC_Modes
       [{\end_ffbd.standby}]})
   StandbyFFBD
   [{\end_ffbd.science}]})
   ScienceFFBD
   [{\set_attitude,sense_attitude,trans_mode,get_modestate}]})
(AttitudeState ||| ModeState))

[CNet]
CNet \ diff(Events,{|cmd_mode,cmd_att,instrument,data|})

-- Consistency check, performed as a test for deadlock
-- We use renaming to to map all events in SCspec to an "abstract_event",
-- and then check for refinement against a terminating deadlock-free
-- process with only abstract_event in its alphabet. This trick seems to
-- reduce the time required for the check to about 80% of the time
-- required for a check using the standard DFtick.
channel abstract_event
DFtick' = (abstract_event -> DFTick') |~| SKIP
SCspec'

SCspec'[ x <- abstract_event | x <- diff(Events,{abstract_event})]]

assert DFTick' [FD= SCspec']

-- Check for our desired property: "downlinked science data should only
-- be collected in valid science attitudes".

-- The Run process
RUN(A) = [] a:A @ a -> RUN(A)

-- Define acceptable science outputs more formally
AcceptableScienceOutput =
{data.(meas,a) | meas <- Measurement,
    a <- {target_attitude_1,target_attitude_2}}
-- The assertion: the only events on channel "data" that SCspec will
-- produce are those in the set AcceptableScienceOutput.
-- We expect this assertion to fail for the unconstrained system, so
-- we prefix the asserted refinement with "not"
assert not RUN(AcceptableScienceOutput) [T=
    SCspec \ diff(Events, {data})

-- If we now assume a "sensible" ground station (i.e. one that won't
-- issue commands to non-science attitudes after the transition to
-- science mode), then the property should be satisfied.
GS = (\a:AttitudeCommand \@ cmd_att!a -> GS)
    \a
    (cmd_att!science_target_1 -> cmd_mode!science -> GS')
GS' = (\a:{science_target_1,science_target_2} \@ cmd_att!a -> GS')

assert RUN(AcceptableScienceOutput) [T=
    (SCspec
        []{{cmd_mode, cmd_att}}]
    GS) \ diff(Events, {data})
Appendix D

Example of Composing Subsystem Models into a Spacecraft System Model

The example presented here defines the behavior of the subsystems for a simple scientific spacecraft, and verifies several different properties of the system model formed by composing those subsystem models. The modeled subsystems are the ADCS, CDH, Communications, EPS, and Payload. The spacecraft is assumed to have no propulsion system, and a passive thermal subsystem, so models for those subsystems are not included in the system model. Furthermore, it is assumed that only the ADCS and Payload have power consumption that varies with their operating mode, and thus only mode power events for those subsystems are included in the model. It is implicitly assumed that the EPS provides enough power in all modes to support the baseline load of the other subsystems, and that the level of available power modeled in the EPS represents power beyond the baseline load level which may be consumed by the ADCS and Payload.

The subsystems have the following characteristics:

- The ADCS behavior is mode-based, and includes a safe-mode transition in response to faults.

- The CDH is primarily a command router, although it also performs some spacecraft initialization tasks, and manages science data collection. It only performs science data collection when the spacecraft is in an attitude suitable for science, and takes a limited number of samples before awaiting a command to commence a new run of sample collection. It responds to ADCS faults by powering down the payload, and transitioning to a safe mode.
• The Communications subsystem consists of simple uplink and downlink behaviors, both modeling the transformation of internal spacecraft signals into RF, and vice versa.

• The EPS is of a type that provides a varying level of power depending on the spacecraft attitude. It also performs some initialization activities in response to the spacecraft separation signal, and manages subsystem power switching and pyrotechnic deployments.

• The payload is a generic scientific instrument. It has two deployable antennas, both of which must be deployed in order to take valid data. The payload captures a sample of data upon command, and transfers it to the CDH subsystem.

In addition to providing an example of constructing a system model from subsystems models, the CSP below is also an example of how early design efforts can help to sharpen a specification as both design and specification evolve in tandem. The original intent of the example was to develop a design which implemented the specification presented in appendix A. However, as the design progressed several areas of the specification were identified which were either under-constrained, or difficult to implement. As a consequence, the specification was revised during the design process to address these issues, with the result that the final specification presented in this example is not the same as the original version that appears in appendix A, but a slightly more complex specification which better reflects the desired behavior. The bulk of the specification remains the same as the original. Most of the modifications consist of additional constraints, which more precisely bound the desired behavior. The major modifications to the specification are:

• All downlink communications now pass through a bounded blocking buffer of depth 1, which ensures consistent handling of downlinked information.

• The transition to safe mode now occurs only after the spacecraft initialization sequence is completed.
• Rather than having fault events directly cause a mode transition, faults now trigger a fault response which includes the mode transition.

• Faults are now permitted to occur during safe mode, as well as nominal mode. However, they are restricted to only occur when the spacecraft is in the \textit{earth\_limb\_scan} attitude state.

• Several new constraints prevent undesirable interactions between the attitude function and the science function, and enforce valid science measurements.

• A new constraint ensures that the downlink of mode status reports is initiated immediately following a mode transition event.

• A new constraint prevents science operations from continuing indefinitely, without any pause to check for incoming commands.

---

```
include "sc-behavior-framework.csp"

-- Type and channel definitions
datatype Attitude = uncontrolled | earth\_limb\_scan | sun\_pointing

datatype AttitudeCommand = detumble | science\_attitude | safe\_attitude

datatype Mode = launch | safe | nominal

datatype CommandMsg = att\_AttitudeCommand | mode\_safe | mode\_nominal


datatype Measurement = sample | junk


nametype ScienceData = (Measurement, AttitudeTlm)

datatype Data = formatted\_ScienceData


datatype DownlinkMsg = modestatus\_Mode | missiondata\_Data | fault\_occurred

datatype Deployables = antenna\_1..2 | solar\_array
```
datatype Subsystem = adcs | cdh | eps | ul | dl | payload

-- External channels
channel uplink : CommandMsg
channel instrument : Measurement
channel downlink : DownlinkMsg
channel separation

-- Specification and implicit interface channels
channel cmdin : CommandMsg
channel attitude : StateIF.Attitude
channel adcs_sense : Attitude
channel in_Sci : ScienceData
channel sci_req, sci_out_ack
channel data : DownlinkMsg
channel trans_Mode : Mode
channel deploy : Deployables
channel fault

-- Subsystem power interfaces
nametype PowerRange = { -10..10}
datatype Power = load_switch.Subsystem.OnOff
| load_delta.Subsystem.PowerRange
channel power : Power
channel power_avail, power_alloc : StateIF.PowerRange
channel power_delta : PowerRange
-- Subsystem command interfaces

datatype PayloadCmd = take_sample

datatype EPSCmd = sw.Subsystem.OnOff | pyro.{antenna.1, antenna.2}

datatype SubsysCmd = adcs_cmd.AttitudeCommand
  | eps_cmd.EPSCmd
  | pl_cmd.PayloadCmd
  | dl_cmd.DownlinkMsg

-- Subsystem telemetry interfaces

nametype AttitudeTlm = union(Attitude, GenericStreamState)

datatype ADCSTlm = att_violation
  | att_cmd_ack
  | att_tlm_stream.StateIF.AttitudeTlm

datatype PayloadTlm = meas.Measurement | tlm_deploy.{1..2}

datatype SubsysTlm = adcs_tlm.ADCSTlm
  | pl_tlm.PayloadTlm

channel systembus : union(SubsysCmd, SubsysTlm)

-- Specification events

nametype ADCSMode = {off, earth_limb_scan, sun_pointing}

nametype PayloadMode = {on, off}

channel modetrans_payload : ModeTransDelimiter.PayloadMode

channel modetrans_adcs : ModeTransDelimiter.ADCSMode

channel eps_exception : ResourceException

-------- Spacecraft system model -------------------------------
SCsystem =
let

INIT_ATTITUDE = uncontrolled
f_ID(x) = x -- identity function

---- ADCS -------------------------------------------------------------
ADCS_IF = {\{power.load_switch.adcs, power.load_delta.adcs,
              systembus.adcs_cmd, systembus.adcs_tlm,
              attitude.trans, fault\}}
ADCS =
let
  -- ADCS functions
  f_ADCSModePower(m) =
    let
      f = {(off, 0),
           (sun_pointing, 4),
           (earth_limb_scan, 5)}
    within apply(f,m)
  f_ACS(c) =
    let
      f = {(detumble, sun_pointing),
           (science_attitude, earth_limb_scan),
           (safe_attitude, sun_pointing)}
    within apply(f,c)

  -- ADCS modes
ADCSMode(off) =
(power.load_switch.adcs.on
  -> modetrans_adcs.begin!sun_pointing
  -> attitude.setval!sun_pointing
  -> modetrans_adcs.end!sun_pointing
  -> ADCSMode(sun_pointing))

ADCSMode(m) =
(power.load_switch.adcs.off
  -> modetrans_adcs.begin!off
  -> attitude.setval!uncontrolled
  -> modetrans_adcs.end!off
  -> ADCSMode(off))

[]
(systembus.adcs_cmd?c
  ->
  let
  m' = f_ACS(c)
  within
  if m' == m
  then systembus.adcs_tlm!att_cmd_ack -> ADCSMode(m)
  else
    (modetrans_adcs.begin!m'
      -> attitude.setval!m'
      -> modetrans_adcs.end!m'
      -> ADCSMode(m')))
\texttt{(m == earth\_limb\_scan) \& (fault -> ADCSfault)}

ADCSfault =

\texttt{-- ignore commands, and try to report attitude violation}
\texttt{(systembus.adcs\_cmd?_ -> ADCSfault)}
\texttt{[]}
\texttt{(systembus.adcs\_tlm!att\_violation}
\texttt{-> modetrans\_adcs\_begin!sun\_pointing}
\texttt{-> attitude\_setval!sun\_pointing}
\texttt{-> modetrans\_adcs\_end!sun\_pointing}
\texttt{-> ADCSMode(sun\_pointing))}

\texttt{AttitudeState = AssignableState(attitude\_setval, attitude\_getval,}
\texttt{attitude\_trans, INIT\_ATTITUDE)}

within
\texttt{((((ADCSMode(off)
}[{\texttt{power\_load\_switch\_adcs}}])}
\texttt{StateTelemetryStream(power\_load\_switch\_adcs, adcs\_sense,}
\texttt{attitude\_trans,}
\texttt{systembus.adcs\_tlm\_att\_tlm\_stream\_setval,}
\texttt{systembus.adcs\_tlm\_att\_tlm\_stream\_getval,}
\texttt{systembus.adcs\_tlm\_att\_tlm\_stream\_trans,}
\texttt{f\_ID})}
\texttt{][{\texttt{modetrans\_adcs}}])}
\texttt{SubsysModePower(off, modetrans\_adcs,}
\texttt{power\_load\_delta\_adcs,}
\texttt{f\_ADCSModePower)) \{modetrans\_adcs\}}}
\texttt{][{\texttt{attitude\_setval, attitude\_trans, adcs\_sense}}]}
AttitudeState
-- Internally use adcs_sense to allow StateTelemetryStream
-- to read the attitude state without blocking other
-- processes that might wish to perform a getval event
[[attitude.getval <- attitude.getval,
  attitude.getval <- adcs_sense]]
  \ {attitude.setval, adcs_sense}|

---- CDH -------------------------------------------------------------

CDH_IF = {power.load_switch.cdh, cmdin, systembus}

CDH =
  let
  Off =
    power.load_switch.cdh.on
    -> systembus.nl_cmd!modestatus.launch
    -> systembus.eps_cmd.sw.adcs.on
    -> AwaitAttitudeAcq

  AwaitAttitudeAcq =
    systembus.adcs_tlm.att_tlm_stream.trans?a
    -> if a == sun_pointing
        then Deployments
        else AwaitAttitudeAcq

  Deployments =
    systembus.eps_cmd.pyro.antenna.1
    -> systembus.pl_tlm.tlm_deploy.1
    -> systembus.eps_cmd.pyro.antenna.2
-- Standby/safe mode

Safe(ready_for_sci) =

((cmdin.mode.nominal

  -> systembus.eps_cmd.sw.payload.on
  -> systembus.dl_cmd!modestatus.nominal
  -> Nominal(ready_for_sci, 5))
[]

(cmdin.mode.safe

  -> systembus.dl_cmd!modestatus.safe
  -> Safe(ready_for_sci))
[]

(cmdin.att?a

  -> systembus.adcs_cmd!a
  -> (((systembus.adcs_tlm.att_tlm_stream.trans?a'

      -> if a' == earth_limb_scan
        then Safe(true)
        else Safe(false))
     []

    systembus.adcs_tlm.att_cmd_ack -> Safe(ready_for_sci)
  []

    systembus.adcs_tlm.att_violation -> HandleADCSFault))
[]

(systembus.adcs_tlm.att_violation -> HandleADCSFault))
-- Science mode
-- * takes science data only when in science attitude
-- * coordinates science data-take
-- * takes no more than 5 consecutive samples before stopping to wait for a command to continue
-- * will not act on attitude commands during a data-take
-- * waits for ack to ensure attitude shift has taken place

Nominal(ready_for_sci, n) =
  (cmdin.mode.nominal
    -> systembus.dl_cmd!modestatus.nominal
    -> Nominal(ready_for_sci, 5))
  []
  (cmdin.mode.safe
    -> systembus.eps_cmd.sw.payload.off
    -> systembus.dl_cmd!modestatus.safe
    -> Safe(ready_for_sci))
  []
  (cmdin.att?a
    -> systembus.adcs_cmd!a
    -> ((systembus.adcs_tlm.att_tlm_stream.trans?a'
      -> if a' == earth_limb_scan
        then Nominal(true, n)
        else Nominal(false, n))
     []
     (systembus.adcs_tlm.att_cmd_ack
      -> Nominal(ready_for_sci, n))
     []
     (systembus.adcs_tlm.attViolation
-> HandleADCSFault)))

[]

(systembus.adcs_tlm.att_violation -> HandleADCSFault)

[]

((ready_for_sci == true) and (n > 0) &

systembus.pl_cmd.take_sample
   -> systembus.pl_tlm.meas?m
   -> systembus.adcs_tlm.att_tlm_stream.getval?a
   -> systembus.dl_cmd!missiondata.formatted.(m,a)
   -> Nominal(ready_for_sci, n-1))

HandleADCSFault =

let

FaultResponse =
   systembus.eps_cmd.sw.payload.off
      -> systembus.dl_cmd!fault_occurred
      -> AwaitSafeAttitude

AwaitSafeAttitude =
   systembus.adcs_tlm.att_tlm_stream.trans?a
      -> if a == sun_pointing
           then
               (systembus.dl_cmd!modestatus.safe
                  -> Safe(false))
           else AwaitSafeAttitude
   within

   FaultResponse
within

Off

---- Comm -------------------------------------------------------------

Comm_IF = {\{cmdin, power.load_switch.ul, downlink,
           systembus.dl_cmd, power.load_switch.dl\}}

Comm =
let
  Uplink =
  let
    UplinkOff = power.load_switch.ul.on \rightarrow UplinkOn

    UplinkOn =
      power.load_switch.ul.off \rightarrow UplinkOff
      []
      cmdin?\_ \rightarrow UplinkOn
  within
    UplinkOff

    Downlink = SwitchedLiftF(systembus.dl_cmd,
                              downlink, f_ID,
                              power.load_switch.dl)
  within
    Uplink || Downlink

---- EPS -----------------------------------------------------------------

EPS_IF = {\{separation, power, systembus.eps_cmd, attitude.trans,
            deploy, qr_exception, eps_exception\}}
EPS =
let
  -- Init sequence
  -- EPS detects separation,
  -- powers up CDH,
  -- autonomously deploys array
  Init =
    separation
    -> power.load_switch.ul.on
    -> power.load_switch.dl.on
    -> power.load_switch.cdh.on
    -> deploy.solar_array -> SKIP

CommandLogic =
  systembus.eps_cmd.sw?x -> power.load_switch!x -> CommandLogic
  []
  systembus.eps_cmd.pyro?x -> deploy!x -> CommandLogic

  -- Available power as a function of attitude state
  MAXPOWER = 10
  available(uncontrolled) = 8
  available(sun_pointing) = 10
  available(earth_limb_scan) = 10

DynamicCapacityCheck =
  let
    Check(pA, pL) =
      if pA < pL
then eps_exception.resource_overflow -> STOP
else DynamicCapacityCheck
within
(power_alloc.trans?pL
  -> (power_avail.getval?pA -> Check(pA, pL)
      []
      power_avail.trans?pA -> Check(pA, pL)))[]
]
(power_avail.trans?pA
  -> (power_alloc.getval?pL -> Check(pA, pL)
      []
      power_alloc.trans?pL -> Check(pA, pL)))[]

AvailablePower(a) =
  (attitude.trans?a’
   -> power_avail.trans!available(a’) -> AvailablePower(a’))[]

power_avail.getval!available(a) -> AvailablePower(a)

AllocatedPower =
  QuantResource(power_delta, power_alloc.getval,
                power_alloc.trans, 0, MAXPOWER, 0)

PowerSource =
  (AllocatedPower
   [|{|power_alloc|}][]
  DynamicCapacityCheck)
  [|{|power_avail|}]
AvailablePower(INIT_ATTITUDE))

{\{power_avail, power_alloc\}}

within

-- Must complete init sequence before anything else happens
Init; (CommandLogic ||| PowerSource)
[[[power_delta <- power.load_delta.s | s <- {payload, adcs}]])

----- Payload -----------------------------------------------
Payload_IF = {\{power.load_switch.payload, power.load_delta.payload,
             systembus.pl_cmd, instrument, systembus.pl_tlm,
             deploy.antenna\}}

Payload =
let
  f_PayloadModePower(m) =
    let
      f = {(off, 0), (on, 5)}
    within apply(f, m)

PLMode(off) =
  (power.load_switch.payload.on
   -> modetrans_payload.begin.on
   -> modetrans_payload.end.on
   -> PLMode(on))

[]
  (power.load_switch.payload.off -> PLMode(off))

PLMode(on) =
  (power.load_switch.payload.off
-- Both antennas must be deployed for useful measurements
-- to be gathered
Antenna(deployed) =
    (card(deployed) == 2) & instrument!sample -> Antenna(deployed)
[]
not (card(deployed) == 2) &
    ((instrument!junk -> Antenna(deployed))
[]
    (deploy.antenna?x
       -> systembus.pl_tlm.tlm_deploy!x
       -> Antenna(union(deployed,{x}))))
within
    ((PLMode(off)
       [[{{instrument|}}]]
       Antenna({})
       [[{{modetrans_payload|}}]]
       SubsysModePower(off, modetrans_payload,
        power.load_delta.payload,
        f_PayloadModePower)) \ {{modetrans_payload|}}
Subsystems = \{(CDH_IF, CDH),
  (EPS_IF, EPS),
  (ADCS_IF, ADCS),
  (Payload_IF, Payload),
  (Comm_IF, Comm)\} within

(|| (IF, Subsys):Subsystems @ [IF] Subsys)
[[cmdin <- uplink]]

InternalChannels = \{|power, cmdin, systembus, attitude|\}
SCsystem' = SCsystem \ InternalChannels

-------- Property verification -----------------------------------------

-- Auxiliary processes RUN, DF (deadlock free), and
df tick (deadlock free, can successfully terminate)
RUN(A) = [] a:A @ a -> RUN(A)
DF(A) = \|\| a:A @ a -> DF(A)
Df tick = \|\| e:Events @ e -> Df tick) \|\| SKIP

Faults = \{|fault, qr_exception, eps_exception|\}

---- Sanity Checks ----

-- PROPERTY: Free of livelock
assert SCsystem' \ union(Faults, {|deploy|}) : [ divergence free ]

-- PROPERTY: Deadlock-free operation does not rely on faults or errors
assert DFtick [F=

((SCsystem' [|Faults|] CHAOS(Faults)) \ Faults)

---- Safety Properties ----

-- PROPERTY: No resource overflows or underflows
assert STOP [T= SCsystem \ diff(Events, {|qr_exception, eps_exception|})]

-- PROPERTY: No instrument use in undesirable attitudes
GoodAtt = {attitude.trans.earth_limb_scan}
BadAtt = diff({|attitude.trans|}, GoodAtt)
assert Between(GoodAtt, BadAtt, {|instrument.sample|}) [T=

SCsystem \ diff(Events, {|attitude.trans, instrument|})]

-- Define desirable science outputs more formally
GoodScienceOutput =

{downlink.missiondata.formatted.(m,a) | m <- {sample},
 a <- {earth_limb_scan}}

BadScienceOutput = diff({|downlink.missiondata|}, GoodScienceOutput)

-- PROPERTY: No downlink of bad science data
assert STOP [T= SCsystem \ diff(Events, BadScienceOutput)]

-- PROPERTY: No downlinking occurs prior to separation from
-- the launch vehicle
SepPrecedesDownlink = separation -> RUN({|downlink|})
assert SepPrecedesDownlink [T=
    SCsystem \ diff(Events, {|separation,downlink|})

---- Liveness Properties ----

-- Signal event for successful completion of a check
channel success

-- PROPERTY: Valid scenario (assuming no faults)
ScenIF = {|separation,downlink,uplink|}
Scenario =
    EventSeq(<separation,
        downlink.modestatus.launch,
        downlink.modestatus.safe,
        uplink.mode.nominal,
        downlink.modestatus.nominal,
        uplink.att.science_attitude,
        downlink.missiondata.formatted.(sample,earth_limb_scan)>)

assert (success -> STOP) [FD=
    ((Scenario; success -> STOP)
    [|ScenIF|]
    ((SCsystem' [|Faults|] STOP)))
    \ diff(Events, {success})

-- EXAMPLE: Invalid scenario (spacecraft must be in science attitude
to take a sample)
InvalidScenario =
    EventSeq(<separation,
        downlink.modestatus.launch,
        downlink.modestatus.safe,
        uplink.mode.nominal,
        downlink.modestatus.nominal,
        downlink.missiondata.formatted.(sample, earth_limb_scan)>)

assert not (success -> STOP) [FD=
    ((InvalidScenario; success -> STOP)
    \|ScenIF\]
    ((SCsystem' \|Faults\] STOP)))
    \ diff(Events, \{success\})

-- PROPERTY: Spacecraft is able to attain all controlled
-- attitude states, in both Safe and Nominal modes
ControlledAtt = \{sun_pointing, earth_limb_scan\}
AttTrans =
    \{|attitude.trans.sun_pointing, attitude.trans.earth_limb_scan|\}

Assumption =
    let
        -- May be in either safe mode or nominal mode
        ModeCmdAssumption = ((uplink.mode.nominal -> STOP) \|~| STOP)
    
        -- Every attitude command is sent
        AttCmdAssumption({}) = STOP
\[ \text{AttCmdAssumption}(\text{Cmds}) = \]
\[ |\sim| \text{cmd:Cmds} @ \text{uplink.att.cmd} \]
\[ \rightarrow \text{AttCmdAssumption}(\text{diff}(	ext{Cmds},\{\text{cmd}\})) \]
\[ \text{within} \]
\[ \text{ModeCmdAssumption} \ ||\ | \text{AttCmdAssumption}(\text{AttitudeCommand}) \]

-- Every controlled attitude is achieved at least once

Commitment =

let

\[ \text{AttCommit}(\{\}) = \text{CHAOS}(\text{AttTrans}) \]
\[ \text{AttCommit}(\text{Atts}) = \]
\[ |\sim| \text{a:Atts} @ \text{attitude.trans.a} \rightarrow \text{AttCommit}(\text{diff}(\text{Atts},\{\text{a}\})) \]
\[ \text{within} \]
\[ \text{AttCommit}(\text{ControlledAtt}) \]

assert Commitment [FD=
\[ (\text{SCsystem} \ [[|\uplink|\}]] \text{Assumption}) \ \text{\textbackslash diff} (\text{Events}, \text{AttTrans}) \]

-- PROPERTY: Produces desirable science

-- Assumptions:
-- * Initially receives command into science attitude and science mode
-- * Consistently receives commands to start a new sample run
-- * On detection of a fault, re-enters science mode
-- * The number of faults is bounded (i.e. cannot diverge on faults)

\[ \text{CmdAssumption} = \]

let

\[ \text{RunSampling}(n) = \]
(n <= 0) & uplink.mode.nominal -> RunSampling(5) 
[]
(n > 0) & downlink.missiondata?_ -> RunSampling(n-1) 
[]
downlink.fault_occurred -> CmdAssumption

within
uplink.mode.nominal
 -> uplink.att.science_attitude
 -> RunSampling(5)

FaultHypothesis =

let
FH(0) = STOP
FH(n) = (fault -> FH(n-1)) |~| STOP
within
FH(3)

assert DF(GoodScienceOutput) [FD=

((SCsystem'
  [[{uplink,downlink.fault_occurred,downlink.missiondata}]])
  CmdAssumption)
[[{fault}]] FaultHypothesis)
  \ diff(Events, GoodScienceOutput)

-- PROPERTY: Always accepts all commands
assert RUN({|uplink|}) [FD=

  ((SCsystem [|Faults|] CHAOS(Faults)) \ Faults )
  \ diff(Events, {|uplink|})
channel init_complete
channel fault_response
SCspec =
  let
    InternalChannels =
      {|in_Sci, sci_req, sci_out_ack, trans_Mode, attitude,
       init_complete, data, fault_response|}

-- Separation behavior
Initialization =
  EventSeq(<deploy.solar_array, attitude.setval.sun_pointing,
             deploy.antenna.1, deploy.antenna.2, init_complete>)

SeparationBehavior =
  EventTrigger({separation}, Initialization)

-- Spacecraft functions
f_Att(c) =
  let
    f = {(detumble, sun_pointing),
         (science_attitude, earth_limb_scan),
         (safe_attitude, sun_pointing)}
  within apply(f,c)
f_Sci((m,a)) = formatted.(m,a)

-- Lifted spacecraft functions
AttitudeCommanding = LiftF(uplink.att, attitude.setval, f_Att)

Science’ = (SeqIn2(sci_req, instrument, attitude.getval, in_Sci)

MIMOLiftF(in_Sci, data.missiondata, sci_req,
                      sci_out_ack, f_Sci))

-- Fault response
FaultResponse =
    EventTrigger({fault},
        EventSeq(<attitude.setval.sun_pointing,
                        fault_response>));
        FaultResponse

FaultReport =
    (trans_Mode.safe -> FaultReport)
    []
    (fault_response
        -> data.fault_occurred
        -> trans_Mode.safe
        -> FaultReport)

-- Spacecraft mode transition behavior
SC_Modes =
    let
TransitionDefs = {(launch, init_complete, safe),
    (safe, uplink.mode.safe, safe),
    (safe, uplink.mode.nominal, nominal),
    (safe, fault_response, safe),
    (nominal, uplink.mode.safe, safe),
    (nominal, uplink.mode.nominal, nominal),
    (nominal, fault_response, safe)}

within

StateTransitions(launch, trans_mode, TransitionDefs)

f_ReportMode(m) = modestatus.m
SC_Modes' = (SC_Modes
    \[\{\{\text{trans_mode}\}\}\]
    LiftF(trans_mode, data, f_ReportMode))
    \[\{\{\text{fault_response,trans_mode.safe}\}\}\]
FaultReport

-- States
AttitudeState =
    AssignableState(attitude.setval, attitude.getval,
        attitude.trans, uncontrolled)

-- Mode constraints
aAtt_Modes = \{|\uplink.att, trans_mode|\}
Att_Modes =
    ModeConstraint(\{|\uplink.att|\}, trans_mode,
        \{|safe,nominal|\}, \{launch\})
aScience_Modes = {\{sci\_req, trans\_Mode\}}
Science\_Modes =
    ModeConstraint({\{sci\_req\}}, trans\_Mode, {nominal}, {launch, safe})

-- Other constraints
aNoAttChange = {\{instrument, data\_missiondata, attitude\_setval\}}
NoAttChangeDuringScience =
    Outside({\{instrument\}}, {\{data\_missiondata\}},
            {\{attitude\_setval\}})

aNoDownlinkBeforeSep = {\{downlink, separation\}}
NoDownlinkBeforeSep = Between({separation}, {}, {\{downlink\}})

-- New constraints
aSciOnlyInSciAtt = {\{sci\_req, attitude\_setval\}}
SciOnlyInSciAtt =
    Between({attitude\_setval.earth\_limb\_scan},
            diff({\{attitude\_setval\}},
                 {attitude\_setval.earth\_limb\_scan}),
            {\{sci\_req\}})

aNoAttCmd = {\{uplink\_mode, init\_complete, fault\_response, data\_modestatus, uplink\_att\}}
NoAttCmdDuringModeTrans =
    Outside({\{uplink\_mode, init\_complete, fault\_response\}},
            {\{data\_modestatus\}}, {\{uplink\_att\}})

aNoSciMode = {\{uplink\_mode, init\_complete,}
fault_response, data.modestatus, sci_req\}

\[\text{NoSciDuringModeTrans} =\]
Outside({|uplink.mode, init_complete, fault_response|},
{|data.modestatus|},{|sci_req|})

\[\text{aNoCmdSci} = \{|sci_req, data.missiondata, uplink|\}\]

\[\text{NoCmdDuringScience} =\]
Outside({|sci_req|},{|data.missiondata|},{|uplink|})

\[\text{aFaultResponseWaitsForSci} =\]
{|sci_req, data.missiondata, fault_response|}

\[\text{FaultResponseWaitsForSci} =\]
Outside({|sci_req|},{|data.missiondata|},{|fault_response|})

\[\text{aAtomicModeStatusReport} =\]
{|separation, init_complete, fault_response, uplink.mode, data.modestatus|}

\[\text{AtomicModeStatusReport} =\]
[] e:{|separation, init_complete, fault_response, uplink.mode|}
@ e -> data.modestatus?_ -> AtomicModeStatusReport

\[\text{aFaultOnlyInSciAtt} = \{|fault, attitude.setval|\}\]

\[\text{FaultOnlyInSciAtt} =\]
Between({attitude.setval.earth_limb_scan},
diff({|attitude.setval|},
{|attitude.setval.earth_limb_scan|},
{|fault|})
\[
\text{aNoInfiniteSci = \{uplink.mode.nominal, instrument\}}
\]

\[\text{NoInfiniteSci = NoInfiniteSci'}(0)\]

\[\text{NoInfiniteSci'}(0) = \text{uplink.mode.nominal} \rightarrow \text{NoInfiniteSci'}(5)\]

\[\text{NoInfiniteSci'}(n) = \text{uplink.mode.nominal} \rightarrow \text{NoInfiniteSci'}(5)\]

\[
\text{[]}
\]

\[\text{instrument?'_ \rightarrow \text{NoInfiniteSci'}(n-1)}\]

\[
\text{aGoodScience = \{instrument\}}
\]

\[\text{GoodScience = instrument.sample} \rightarrow \text{GoodScience}\]

\[-- \text{Constraint network}\]

\[\text{ConstraintSet =}\]

\[
\{(a\text{SciOnlyInSciAtt,SciOnlyInSciAtt}),
(a\text{Att_Modes,Att_Modes}),
(a\text{Science_Modes,Science_Modes}),
(a\text{NoAttChange,NoAttChangeDuringScience}),
(a\text{NoAttCmd,NoAttCmdDuringModeTrans}),
(a\text{NoSciMode,NoSciDuringModeTrans}),
(a\text{NoDownlinkBeforeSep,NoDownlinkBeforeSep}),
(a\text{NoCmdSci,NoCmdDuringScience}),
(a\text{FaultResponseWaitsForSci,FaultResponseWaitsForSci}),
(a\text{AtomicModeStatusReport,AtomicModeStatusReport}),
(a\text{FaultOnlyInSciAtt,FaultOnlyInSciAtt}),
(a\text{NoInfiniteSci,NoInfiniteSci}),
(a\text{GoodScience,GoodScience})\}
\]

\[\text{aCNet = aConstraintNet(ConstraintSet)}\]

\[\text{CNet = ConstraintNet(ConstraintSet)}\]
within

((((((AttitudeCommanding
    ||| Science’)
    ||| SC_Modes’)
    [{init_complete}]
SeparationBehavior)
    [{fault_response}]
FaultResponse))
    [{attitude.setval, attitude.getval}]
AttitudeState)
    [{data}]
BoundedBlockingBuffer(data, downlink, 1))
    [aCNet]
CNet)
  \ InternalChannels

-- The design does not do anything forbidden by the specification
assert SCspec [T= SCsystem’]

-- The design does the things the specification requires it to do
assert SCspec [F= SCsystem’]

-- The specification/design correspondence does not rely on faults
assert SCspec [|Faults|] CHAOS(Faults) [F=
                      SCsystem’ [|Faults|] CHAOS(Faults)
Vita

Allan McInnes

Education

- Ph.D. Electrical and Computer Engineering (Phi Kappa Phi), Utah State University, Logan, UT, 2007.
- M.S. Aeronautics and Astronautics, Purdue University, West Lafayette, IN, 2000
- B.E. Electrical and Electronic Engineering (1st Class Honours), University Of Canterbury, Christchurch, New Zealand, 1997

Professional Experience

- Embedded System Engineer, Syncroness, Inc. Westminster, CO, Oct. 2006 - Present

Honors and Awards

- Space Dynamics Laboratory Tomorrow Ph.D. Fellowship, 2003-2006
- Aerospace Corporation Spot Award (GPS III), 2002
- Aerospace Corporation Performance Recognition Award (ST5), 2002
• Aerospace Corporation Performance Recognition Award (MER), 2001
• Aerospace Corporation Achievement Award (Mars Exploration Rover), 2001
• Purdue University Frederick N. Andrews Fellowship, 1998-2000
• Purdue University Warren G. Koerner Fellowship, 1998-2000
• Canterbury Research Award (Masters Level), 1997

Publications

Conference Proceedings


Technical Reports


Invited Talks


Significant Professional Presentations

• A. I. McInnes, “Developing Concurrent Design Teams,” Presentation to NASA Ames Research Center, Feb. 6th, 2003

• A. I. McInnes, “STSS (SBIRS Low) Common Bus Study Final Results,” Briefing to STSS Core Team, Dec. 13, 2002

• A. I. McInnes, “Concept Design Center Space Segment Team Support for GPS III,” Presentation to Aerospace Corporation Engineering Technology Group Vice President, Sep. 17, 2002

• S. M. Feldman, A. McInnes, “SBIRS Low Common Bus Study Results,” Briefing to SBIRS Low Program Office, Jun. 4, 2002

• A. I. McInnes, D. A. Bearden, P. B. Burridge, “New Millennium Program ST5 COTS Spacecraft Compatibility Assessment,” Briefing to NASA HQ and New Millennium Program Managers, May 8, 2002